MASTER IN COMPUTER SCIENCE

From organizational to system requirements the debit card purchase case study

Backes, Constant

Award date:
1997

Awarding institution:
University of Namur

From Organizational to System Requirements :

The Debit Card Purchase Case Study

Constant Backes

Promoter : Prof. Eric Dubois

Thesis submitted in conformity with the requirements for the degree of 'Licencié et Maître en Informatique'
1996-1997

RUE GRANDGAGNAGE, 21 **B - 5000 NAMUR (BELGIUM)**

Acknowledgements

First of all I want to thank Eric Dubois and his team for their help and their precious advices. Especially I want to express my gratitude to Phillipe Du Bois for never loosing his patience. Thanks also to Michael Petit for helping me with some Albert issues.

I am also grateful to John Mylopoulos and Eric Yu from the University of Toronto for providing me an ideal environment for doing my research.

Thanks to my parents for their financial support. My gratitude also goes to my sister Claudine and her husband Armand for their help and assistance during these long years.

I have also to mention Chuck, Carole and Penny Cookson who have helped me to forget the cold Toronto's nights.

Finally, I owe my deepest gratitude to my wife Denise for her moral support.

Abstract

The improvement of existing systems, organizations or processes becomes more and more difficult. This is mainly due to the fact that their implementation is often based on specifications describing what has to be achieved by the different entities. The reasons why a given agent is or has to act in a predefined way are however not represented.

The aim of this work is to present a possible way to overcome this problem. The described solution consists in combining two specification frameworks : the Albert II language and the i* framework.

The Albert II language, developed at the Facultés Universitaires Notre Dame de la Paix in Namur, allows us to represent what has to be achieved by the agents of a given system. The i* framework developed at the University of Toronto, at its turn, allows us to express why a certain agent is acting in the way it does.

The aim of this work also consists in describing the existing links between both frameworks, how the two frameworks may be used together and what are the analyst's advantages of using both frameworks.

Abstrait

L'amélioration de systèmes, d'organisations ou de processus existants devient de plus en plus difficile. Ceci est surtout dû au fait que leur implementation est le plus souvent basée sur des spécifications decrivant ce qui doit être fait par les différents entités. Les raisons pourquoi un agent agit ou doit agir selon une manière bien précise doivent être connues et prises en compte afin d'améliorer un système donné.

Le but de ce travail consiste à décrire une approche qui permet de résoudre ce problème. La solution consiste à utiliser conjointement deux langages de spécification : le language Albert II et le language i*.

Le language Albert II a été développé aux Facultés Universitaires Notre Dame de la Paix à Namur et nous permet de décrire ce qui doit être réalisé par les agents d'un système donné. Le language i*, développé à l'Université de Toronto, permet d'exprimer les raisons pour lesquelles les differents agents agissent ou doivent agir d'une manière précise.

Le but de ce travail consiste également à décrire les liens existant entre ces deux langages, comment les deux langages peuvent être utilisés conjointement et quels sont les bénéfices qu'un analyste peut en tirer.

CONTENTS

TABLE OF FIGURES

Introduction

Whereas the development of a complex system still represents an interesting and challenging task for an analyst, the improvement of a system may become a difficult and even impossible task. This is mainly due to the fact that the documents the analyst has at its disposal, in order to improve an existing system, often only describe what has to be realized by the different entities of the system.

In order to improve a given system, a deeper understanding of the system and the context in which the system is embedded is however required. This understanding is obtained by analyzing the reasons why a given agent is or has to act in a predefined way. Most specification languages do however not allow the representation and description of those *Whys*.

In this paper, we describe a way that allows the analyst to specify both the *Whats* and the ***Whys*** of a given system. The proposed way consists in combining two different existing specification languages : the Albert II language and the i* framework.

The Albert II language is a formal requirements specification language which can be used in order to describe what has to be realized by the different entities of a given system. The Albert II language has been developped at the Facultés Notre Dame de la Paix in Namur, (Belgium) and suits particularly for representing real-time distributed and cooperative systems.

"Basically, Albert is based on a variant of real-time temporal logic, a mathematical language particularly suited for describing histories (i.e. sequences of states) and expressing performances constraints " [1].

The second framework we use is the i* framework developed at the University of Toronto, Ontario (Canada). The i* framework allows us to represent the *WHYs* of a given organization or system by taking an agent-orientated approach.

The i* framework allows us to represent a given organization or system by describing the existing agents as well as the dependencies which exist between them. It also allows to represent the behaviour of the different agents i.e. the behavior the different agents have to adopt in order to produce, achieve or execute the source of the dependency, called dependum at the i* level.

The particularity of the i* framework consists in the fact that the different agents are perceived by the i* framework as an intentional and strategical entity. This means that the i* framework recognizes and allows to represent the actors' desires, wants and goals. An agent is considered as a strategical entity as it is concerned about its opportunities and vulnerabilities in the relationship with the other agents. The notion of *softgoal* is used to describe a non-sharply defined goal of an actor.

Another characteristic of the i* framework consists in allowing the representation of different alternative ways which allow a particular agent to bring about the same dependum. As different alternatives may have positive or negative implications for a given actor, a special notation is used in order to represent those implications.

By describing the *Whats* and *Whys* of a given organization or system, the analyst's task is simplified in the case where a particular organization or system has to be improved. The use of both frameworks however also allows to facilitate the specification of a new system due to the *natural* link that exists between both frameworks.

The aim of this paper consists in describing the existing link between both frameworks, the way both frameworks may be used together as well as the advantages of the proposed approach.

Chapter 1. Introduction to the Albert II language

The Albert language is composed of two different components : the Declaration component and the Constraints component.

The Declaration component allows us, by using graphical notations, to describe the vocabulary of an application or system. The vocabulary consists in the declaration of the different involved agents, their internal states, the actions that can be executed by them as well as their cooperation.

The Constraints component, represented by a set of formal logical statements, expresses the constraints that the different agents have to respect during their life-cycle. Whereas the declaration part only informs us that a certain agent may for instance execute a certain action, possesses a certain number of internal states and cooperates with other agents, the constraints component allows us to specify the circumstances under which an action has to be executed as well as the effects that the execution of that particular action has on the different agent's state components. It also allows us to restrict the evolution of the different internal states of an agent by using logical and temporal expressions. Finally, the circumstances under which agents can or have to cooperate can also be specified.

In this chapter, we illustrate the use of the Albert II language through the handling of a case study which has previously been introduced by Philippe Du Bois. The description of our example called the *Four Messages Protocol* example is given in Section 1. Section 2 and Section 3 describe the Declaration respectively the Constraints component. Each Albert notion is illustrated by one or several examples based on the *Four Messages Protocol* example. The final obtained Albert specification is described in Section 4.

As the aim of this chapter consists only in giving a short introduction to the Albert II language, we inform the reader that a detailed description of the Albert language can be found in [2] and [3] which have been the foundation of this chapter.

Section 1. The Four Messages Protocol Example

Description of the example :

A customer applies to a shopkeeper in order to buy a certain number of items. We assume that the customer pays these items by using its debit card and that a C-ZAM system is installed in the shop.

The aim of the C-ZAM system is to allow customers to pay for goods or services by using their debit card. From the customer's viewpoint, a certain number of information are entered into the terminal located in the shop. These information are then forwarded by the terminal to the customer's bank. Depending on the received information, the bank decides whether the transaction request can be accepted or not. Its decision is then sent back to the terminal where it is displayed.

If the customer's transaction request has been accepted by the bank, the price of the transaction is displayed and has to be validated by the customer. The validation of the displayed price consists in pushing the terminal's OK button in the case where the correct transaction amount has been displayed. If an incorrect price has been displayed, the customer has to cancel the transaction process by pushing the CANCEL button. In the case where the customer's transaction request has been refused, the reason of the refusal is displayed.

Let us now analyse the structure of the C-ZAM system in more details :

" The system is composed of a certain number of C-ZAM terminals and of a C-ZAM Host. A C-ZAM terminal is located in the shop and is linked by a telephone line to the C-ZAM Host (the line may be leased or switched). A terminal is composed of a card reader, a keyboard and a single line LCD screen. The terminal has an internal permanent memory and is able to keep data even if the power is turned off. "

The communication between the different agents are submitted to a certain number of rules. These rules are prescribed by a protocol called the Four Messages Protocol (4MP).

The Four Messages Protocol (4MP)

In a Four Messages Protocol, four messages have to be exchanged between the different involved agents before the transaction process is completed. The different exchanged messages are summarized by Figure 1.1..

The transaction process begins by sending an Authorization Request (ARQ) message to the C-ZAM Host. The ARQ message contains three information : (i) the customer's debit card number obtained by scanning the customer's debit card through the terminal's card reader, (ii) the transaction amount and (iii) the customer's secret PIN (Personal Identification Number) code.

Once the ARQ message has been sent to the C-ZAM Host, the terminal waits a certain period of time in order to get a response from the C-ZAM Host. If the Host's answer does not arrive within that period, a Timeout occurs and the transaction process is aborted.

Having received the Authorization Request (ARQ) message from the terminal, the C-ZAM Host verifies if the customer has entered a valid PIN code. Is this the case, the C-ZAM host saves the transaction information into its local database, determinates the customer's bank to which the message has to be forwarded, replaces the customer's debit card number by the customer's account number and forwards the modified transaction request (ARQ) to the bank. The Host then waits a certain period of time in order to get the bank's response. If the bank's response does not arrive within that period, a Timeout occurs at the Host's level and the transaction process is aborted by the Host. A message is then sent to the terminal informing it that the transaction process could not have been completed. If the bank's response arrives at the Host's location after a Timeout occurred, the response is ignored.

If the customer has entered an invalid PIN code, the transaction request is refused by the Host and the request is not forwarded to the customer's bank.

On the basis of the received transaction information (i.e. the customer's account number and the transaction amount), the bank decides whether the transaction request can be accepted or not.

A transaction request is accepted if the customer has enough money on its account in order to cover the transaction. In our example, we say that a transaction is covered if the customer's balance is superior or equal to zero once the transaction amount has been retrieved from the customer's account. In practice however, a customer can spend more than it possesses. In such cases, the bank has to evaluate the customer's financial situation by taking into account several parameters. As these parameters vary from one bank to another and in order to keep our example simple, we assume that a transaction request is only accepted by the bank if the customer's balance is not negative once the transaction amount has been retrieved.

The transaction information as well as the bank's decision are recorded and the bank's response is sent to the C-ZAM Host. The message containing the bank's response is called Authorization / Reject (AUTREJ) message as the customer's transaction request is either AUThorized or REJected by the bank. After the reception of the bank's response, the Host updates its transaction record based on the received information and transfers the received message to the terminal from where the transaction request has been emitted.

If a message arrives at the terminal informing the customer that the transaction request has been refused, the reason of the transaction's refusal is displayed on the terminal's LCD screen. In the case where a Timeout occurred i.e. in the case where no answer is received from the C-ZAM Host during a certain period of time, the transaction is aborted and a message is displayed on the screen informing the customer that the transaction could not have been executed. Otherwise, the transaction price is displayed and has to be validated by the customer.

The customer's confirmation is then sent to the Host in a message called Confirmation (CONF) message. The Host updates its corresponding transaction record and forwards the received confirmation message to the customer's bank. In the case where the customer has validated the displayed amount, the bank is updating the customer's account balance by retrieving the validated amount from the customer's account.

In the *Four Messages Protocol*, we make the distinction between two different categories of exchanged messages. The criteria which has to be applied in order to find out to which category a certain message belongs depends on the consequences that the loss of the message implies.

The first category regroups the messages we call *non critical* messages and includes the Transaction Request (ARQ) as well as the bank's Authorization / Reject (AUTREJ) message. If one of those messages does not arrive at destination (a message is for instance jammed or lost due to a broken line), a Timeout occurs at the Host's or at the terminal's level and the transaction is aborted. If, for instance, the terminal does not get an answer from the Host after a certain while, a message will be displayed on the terminal informing the customer that the transaction process can not be executed and that the process is aborted. The customer may be disappointed but can be assured that no money is removed from its bank account.

The second category of messages are regarded as *critical* messages as the loss of these messages will have a considerable consequence for one or more of the implicated agents. The confirmation (CONF) message is an example of such a critical message.

Let us assume that a customer has validated the transaction price. From the customer's viewpoint, the transaction process is finished. The customer can assume that the transaction request is or will be transferred to its bank and that the right amount of money is or will be retrieved from its account. As a result, the customer is the regular owner of the bought items.

If a problem now occurs that prevents the confirmation message to be delivered to the customer's bank, the transaction amount is never retrieved from the customer's account. In such a situation, the customer may be satisfied as he or she gets the items for free. The shopkeeper may however be less satisfied as he\she may face the risk of getting never refunded from the customer's bank.

As the loss of the exchanged messages is in a direct relation with the quality of the used lines and equipment and as the lines and equipment used by the C-ZAM Host and the Bank are usually of a good quality, the chances that a message will be lost during the transfer between the C-ZAM Host and the Bank are very low. The only problems that may occur during the transfer of the exchanged messages are located between the terminal and the C-ZAM Host as shopkeepers are usually not disposed to install expensive equipment in their shops or lease expensive communication lines.

To prevent the loss of a Confirmation message, we assume that the reception of the Confirmation message has to be confirmed by the Host. We further assume that this acknowledgement of receipt is done by sending a message, called Acknowledge (ACK) message, from the Host to the terminal. As the Acknowledge message also may face the risk of getting lost, we finally assume that the terminal stores the Confirmation message into its internal permanent memory (the internal permanent memory prevents the Confirmation message to be lost) and that the terminal has to send the stored message to the Host in regular intervals as long as it does not receive an Acknowledge message in return.

Analysis of the structure of the exchanged messages

Figure 1.1. describes that four different types of messages have to be exchanged between the different involved agents before the transaction process is terminated. The aim of this sub-section consists in defining the structure i.e. the content of each sent message.

Figure 1.2. describes the basic exchanged information contained in the different exchanged messages. The terminal sends the customer's PIN code, the transaction amount and the customer's debit card number to the C-ZAM Host. After having verified the validity of the customer's PIN code and replaced the customer's debit card number with the customer's account number (in the case where the customer has entered a valid code), the C-ZAM host transfers the information to the customer's bank.

The customer's PIN code, the transaction amount and the debit card respectively the account number represent the basic transaction information of the transaction process and are used by the C-ZAM Host respectively the Bank agent to decide whether the transaction request can be

forwarded respectively can be accepted or not. Additional information have however to be included in the different messages in order to solve problems that may occur if only these basic transaction information are transferred.

The C-ZAM host, for instance, may receive hundreds of requests from different terminals

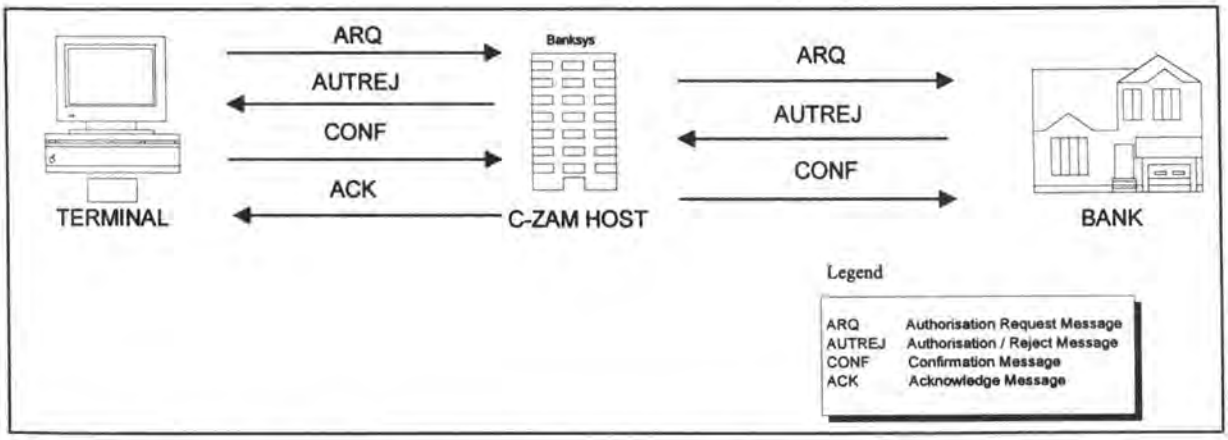


Figure 1.1. Exchange of messages during a regular Transaction Process based on the Four Messages Protocol

during a day. With the information described by Figure 1.2., it is impossible for the C-ZAM Host to find out the source of the received messages. The source of the messages is however needed by the Host in order to forward the bank's response.

In order to solve this kind of problem, we assign a unique identification code to the terminal and bank agent. We further assume that the terminal and bank agent has to include its identification code in each message it forwards to the Host. If the C-ZAM Host receives for instance an ARQ message from the terminal, it can use the terminal's identification code (and compare it to a list of known identification codes) in order to find out the identity and location of the terminal.

A second problem that may occur is that an agent receiving a response from another agent does not know to which transaction request the response belongs. In order to prevent this kind of problems, we prescribe that the basic information (the customer's PIN code, the transaction amount and the customer's debit card respectively account number) have to be included in each sent message. By including these basic information into each message, the size of the messages however increases and the messages become theoretically more vulnerable to transmission errors.

A solution to this problem may consist in prescribing the storage of the received or sent information into a local database respectively into a permanent memory and in forwarding a sort of reference key to those recorded information.

If the Host, for instance, receives a transaction request from a terminal, it stores the received information into its local database and forwards them (after having replaced the customer's debit card number by its account number) as well as a reference key (pointing to the received and stored transaction information) to the customer's bank. The Bank sends its response back

to the Host as well as the Host's reference key. The reference key is then used by the Host in order to find out to which transaction request the response belongs. The transaction information are then updated, and a new message is created and is forwarded to the terminal.

By using this mechanism, the bank's response message, for instance, contains only three

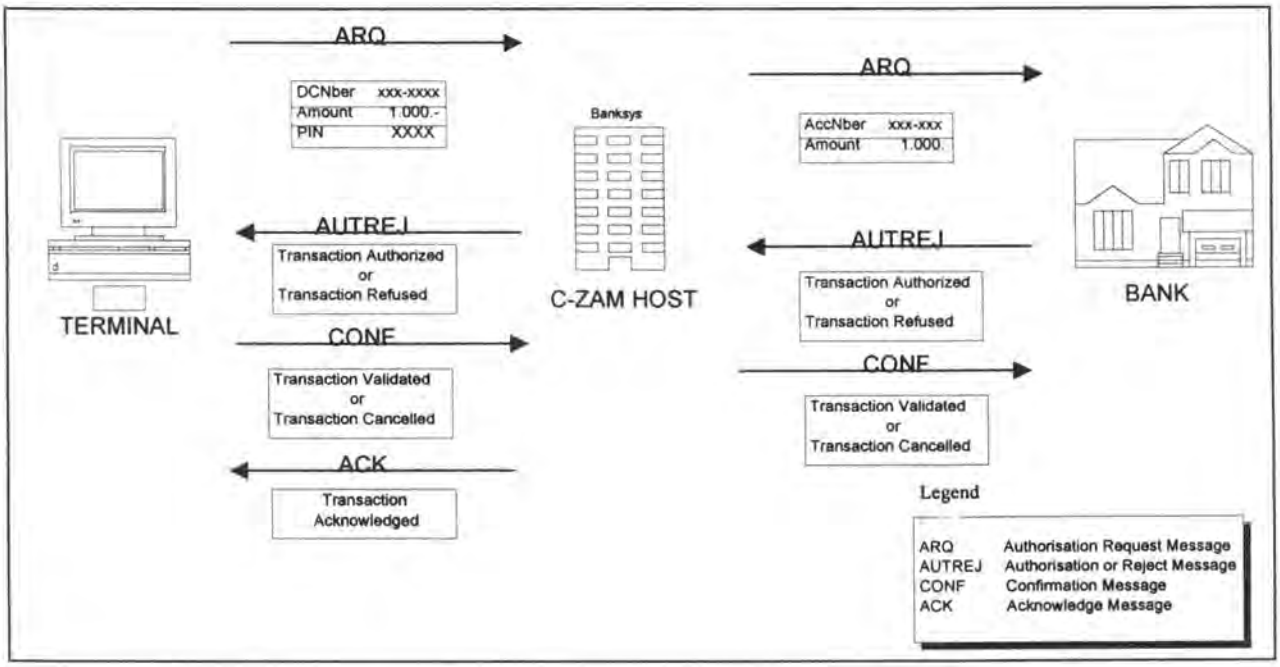


Figure 1.2. Basic Information exchanged between the agents in a Four Messages Protocol Transaction Process

information : the bank's response (whether the transaction request has been accepted or not), the C-ZAM Host's and the bank's reference key. The bank's reference key has to be returned by the Host with the Confirmation message to the bank and is used by the bank to find out to which transaction the received confirmation message belongs.

A similar mechanism can also be used at the terminal's level. We previously assumed that the terminal has to store the Confirmation message into its permanent memory in order to avoid the loss of the message once the power is turned off. As several Confirmation messages can be contained in the terminal's memory, the Host has to specify clearly which Confirmation message it has received from the terminal. Once again, the Host could do this by sending all the received information back to the terminal. The number of exchanged information can however be reduced by integrating into the confirmation message a reference key pointing to the in the terminal's memory stored information. By doing this, the Host has only to sent back the reference key which is used by the terminal to access and remove the sent Confirmation message from its internal permanent memory.

The last problem that may occur is due to the fact that a terminal can emit different transaction requests which may contain the same basic transaction information. In order to make the distinction between those transactions, we assume that a terminal can make only one and one only transaction request at a given moment and we prescribe that two information have to be

added to the basic transaction information : the date and time when the transaction is requested.

The final structure of the different exchanged messages is given by Figure 1.3..

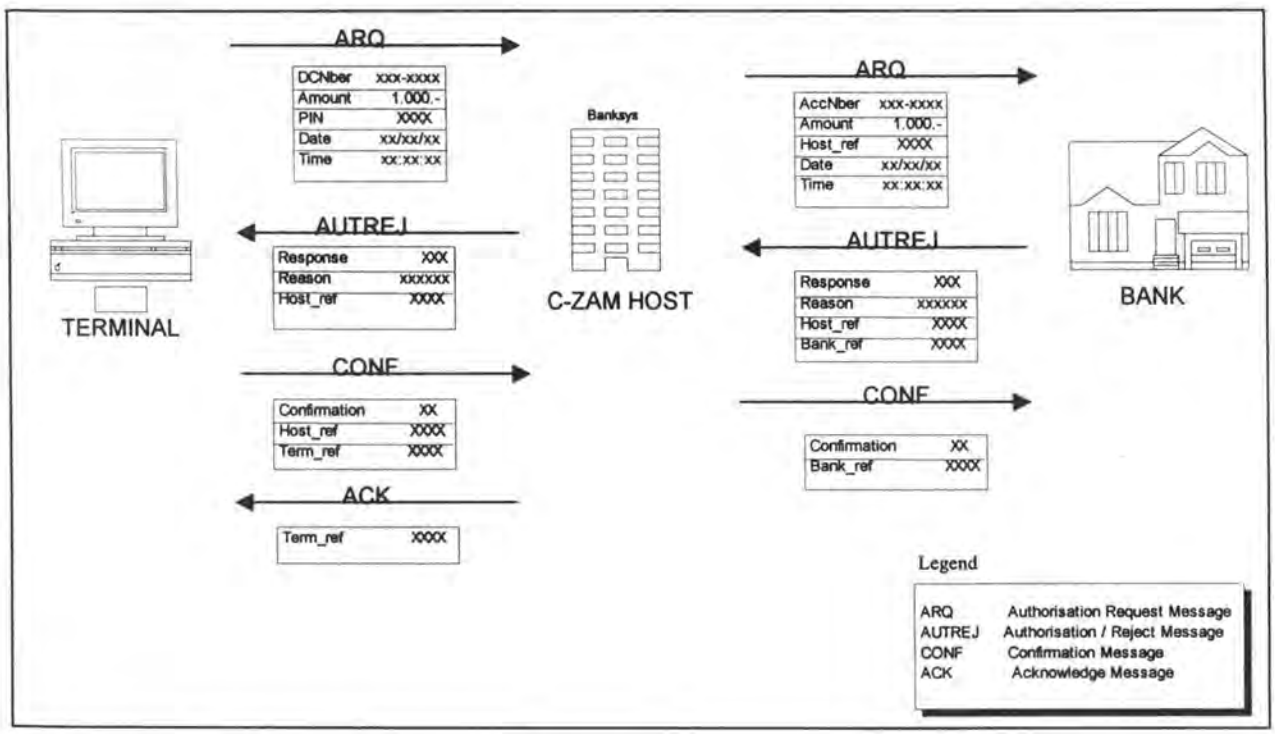


Figure 1.3. Structure of the exchanged messages in the Four Messages Protocol example

Having given the description of the *Four Messages Protocol* example and described the structure of each exchanged message, we now analyse the first of the two Albert components : the Declaration component.

Section 2. The Albert's Declaration Component

The aim of the Declaration component is to define "the structure of the composite system in terms of agents as well as the structure of each individual agent [4] "

The declaration of a society

The structure of the composite system, called society, specifies by using graphical notations, the different agents that are included in the system as well as the number of their occurrences (single agent or member of a class).

In our *Four Messages Protocol* example, different agents can be identified. First, there is the customer which applies to the shopkeeper in order to purchase a certain number of goods or services. The shopkeeper and the customer provide the transactions information to the terminal which forwards them to the C-ZAM Host agent. In the case where the customer has entered a valid PIN code, the request is then forwarded to the customer's bank.

In our *Four Messages Protocol* example, we are mostly interested in the exchange of messages between the different components of the C-ZAM system. Three agents may be identified in the C-ZAM system : the Terminal, the C-ZAM Host and the Bank agent. The so obtained society may then be incorporated as a sub-component into a more complex society including this time the customer and the shopkeeper agents.

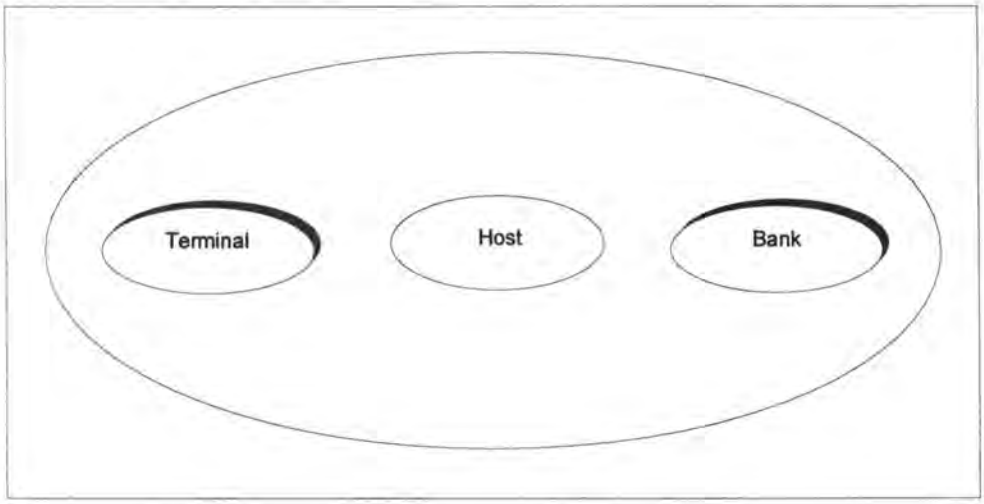


Figure 1.4. Society of the Four Messages Protocol

Graphically, a society is represented by an ellipse containing smaller sub-ellipse components representing the different agents. Among the sub-ellipses, we make the distinction between ellipses with shadows depicting a class of agents and those without shadows depicting single agents.

Figure 1.4. specifies that our *Four Messages Protocol* society is composed of several terminals, one C-ZAM Host and several bank agents.

Let us remark that in the last Albert version, the analyst may specify a goal at the societies' level which has to be respected or achieved by the different members of the society. This goal is expressed by a formula and may refer to an agent's action or state component. How the condition or state is achieved is then described in the declaration of the different agents.

A society represents the different types and occurrences of the involved agents but does not give us information about their interactions, the actions that may be executed by them during their life-cycle or their internal states. Those kind of information may however be obtained by analyzing the declaration of the different agents.

The declaration of an agent

The declaration of an agent is obtained by specifying the internal state components and actions of the agent as well as the agent's cooperation with the other agents.

a. Declaration of the internal state components and actions of an agent

The agent's internal state components as well as the actions that may be executed by the agent are represented inside a parallelogram. Actions are represented by a box containing the action's name and a circle. The circle is used in order to make the distinction between an action and a state component. Parameters can be assigned to actions which affect their execution. The action's parameter are typed and are represented outside the box. They are linked to the box by a line.

In addition, the Albert II language includes the mechanism of decomposing an action into sub-actions. It also introduces the notion of "actions that may not occur outside the context of a composed action". To make the distinction between these and the other action, the actions that may not occur outside the context of a composed action are graphically represented by a bold box.

Each state component is of a certain type. The type of each state component is specified in a box located at the bottom of the state component box. In Albert, we make the distinction between 4 different types :

- *the predefined elementary data types* : the predefined elementary data types are the BOOLEAN, the INTEGER, the RATIONAL, the CHAR, the STRING and the DURATION type. Each data type is accompanied by its usual operations. For a complete list of operations, please consult Appendix B of [2].

- *the elementary types defined by the analyst* : elementary types are defined by the analyst and are based on the characteristics of the system that has to be described or implemented. The only operators that can be used for this kind of type are the equal (=) and the not equal (≠) operators.

- *the constructed data types* : More complex data types can be created by the analyst by using a set of predefined type constructors. The type constructors that can be used in an Albert II specification are the cartesian product (CP), the set (SET), the bag (BAG), the table (TABLE), the union (UNION) and the enumeration (ENUM) constructors.

- *the types corresponding to agent identifiers* : in Albert II, a type is automatically associated to each class of agents. Inside the agent's declaration, the *self* constant is used to refer to the proper identifier of the described agent.

In our *Four Messages Protocol* example, different elementary types have been defined by the analyst. The CARD type is one of those types and is assigned to the customer's debit card number. The Albert specification informs us that the customer's debit card number is of type CARD. Details about the composition of the customer's debit card number like for instance its length is however not given. These details will be specified further on at the implementation level.

An example for the constructed data type is given by the *Transactions* table component (obtained by applying the *table* constructor). The *Transaction* table is used by the Host and

the bank agent in order to store the received transaction information during the different transaction steps.
Figure 1.4. describes the fact that the society of the *Four Messages Protocol* example is composed of three different agents. Automatically, three types corresponding to the agent identifiers are created. These types are the TERMINAL, the HOST and the BANK type.

Different graphical notations are associated to the state components depending on their types. See Figure 1.5. for more details.

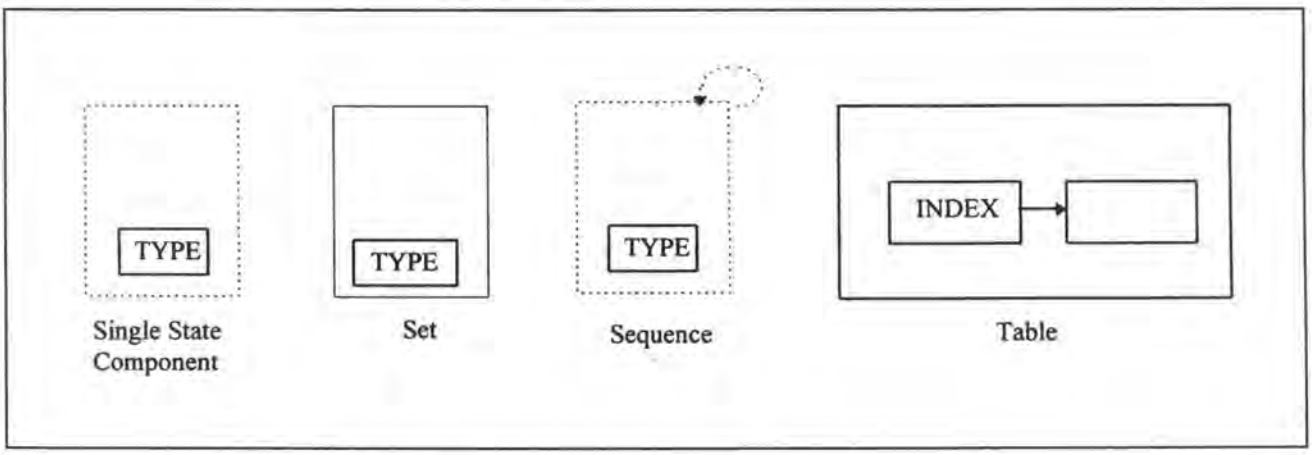


Figure 1.5. Graphical Representation of the different state components

In order to make the distinction between time varying and constant state components, a bold line is used to represent a constant component and a plain line to represent time varying state components.

Figure 1.6. describes the graphical declaration of the Host agent. By analyzing the content of the parallelogram, we can find out that the Host agent possesses a state component called TimeOutPeriod. It is an individual constant component which represents the interval of time during which the Host agent waits in order to get the response from the customer's bank. If the bank's response does not arrive during this period of time, a Timeout occurs and the transaction process is aborted.

Four different table components are defined for the Host agent. The *Transactions* table for instance contains the different transaction records that are recorded during the agent's life-cycle. The transaction information are of type TRANS_H and are indexed by REF_H. The key value (of type REF_H) is a unique value and allows to access information corresponding to a certain transaction. It is also this value that is forwarded as reference key to the other agents.

The Pin table component contains the different debit card numbers emitted by the Host as well as the corresponding PIN codes.

The Albert language also allows us to define state components whose values are derived from other state components. The special link between these components is represented by a wavy line.

In our *Four Messages Protocol* example, the Host receives a transaction request from a terminal. Before the Host forwards the received authorization request message to the customer's bank, it has to find out if the customer has entered a valid PIN code. If we assume that the received debit card number is *dc* and that the received PIN code is *p*, the Host has to execute two checks :

First, it has to verify if the received debit card number is a valid number i.e. if the received number *dc* corresponds to a key value of the PIN table. Second, it has to find out if the entered PIN code is a valid code i.e. if the received PIN code *p* matches the Pin code obtained by accessing the PIN table with the key value *dc*.
If we assume that the result of each check is stored in a state component called *ValidCard* respectively *PinMatches (*both state components are of BOOLEAN type), the final decision whether to forward the customer's request or not, represented by the *ForwardRequest* state component, depends on the value of the two state components *ValidCard* and *PinMatches*.

The wavy line represents the special link between the three state components. How the value of the derived *ForwardRequest* component is calculated is not specified by the graphical description. The derivation rule can however be found in the textual Albert part.

b. The agent's relations with the other agents

The parallelogram's border and the area outside of it describe the potential interaction that an agent may have with the rest of its society. Usually, an agent is not an isolated subject but evolves in a certain environment. In order to achieve difficult tasks, the cooperation between agents is often requested. In Albert, a cooperation consists in communicating information between agents. The communicated information refer either to the value of a certain internal state or to the fact that a certain action has been executed.

A dotted arrow leaving the parallelogram, indicates that an external agent is informed about the value of a certain state component (in the case where the source of the arrow is a state component) or the execution of a certain action (in the case where the source of the arrow is an action). The external agent being informed is specified by the label at the end of the arrow.

An arrow with a state or an action at its end, describes the fact that the value of an external state component or the execution of an external action is communicated to the agent described by the parallelogram. The source of these information is again specified by a label, located this time at the beginning of the arrow.

The Host, for instance, communicates the execution of four actions to the terminal respectively to the bank agent. The four actions can be easily identified by analyzing the dotted arrows leaving the parallelogram depicted by Figure 1.6..

The execution of three external actions is communicated to the Host agent. The agents executing the different actions are the terminal (for the *SendMsgConf* and the *SendMsgReq* actions) respectively the Bank agent (for the *EnvoiMsg* action).

Exporting or importing information however does not signify that the destination agent perceives the state components or actions during its entire life-cycle. The Albert Constraints

part, composed of a set of logical and temporal expressions, allows us to define conditions under which the importation and exportation of actions or state components are perceived or hidden to the destination agent. How such constraints are expressed is analysed, among others, in the following section.

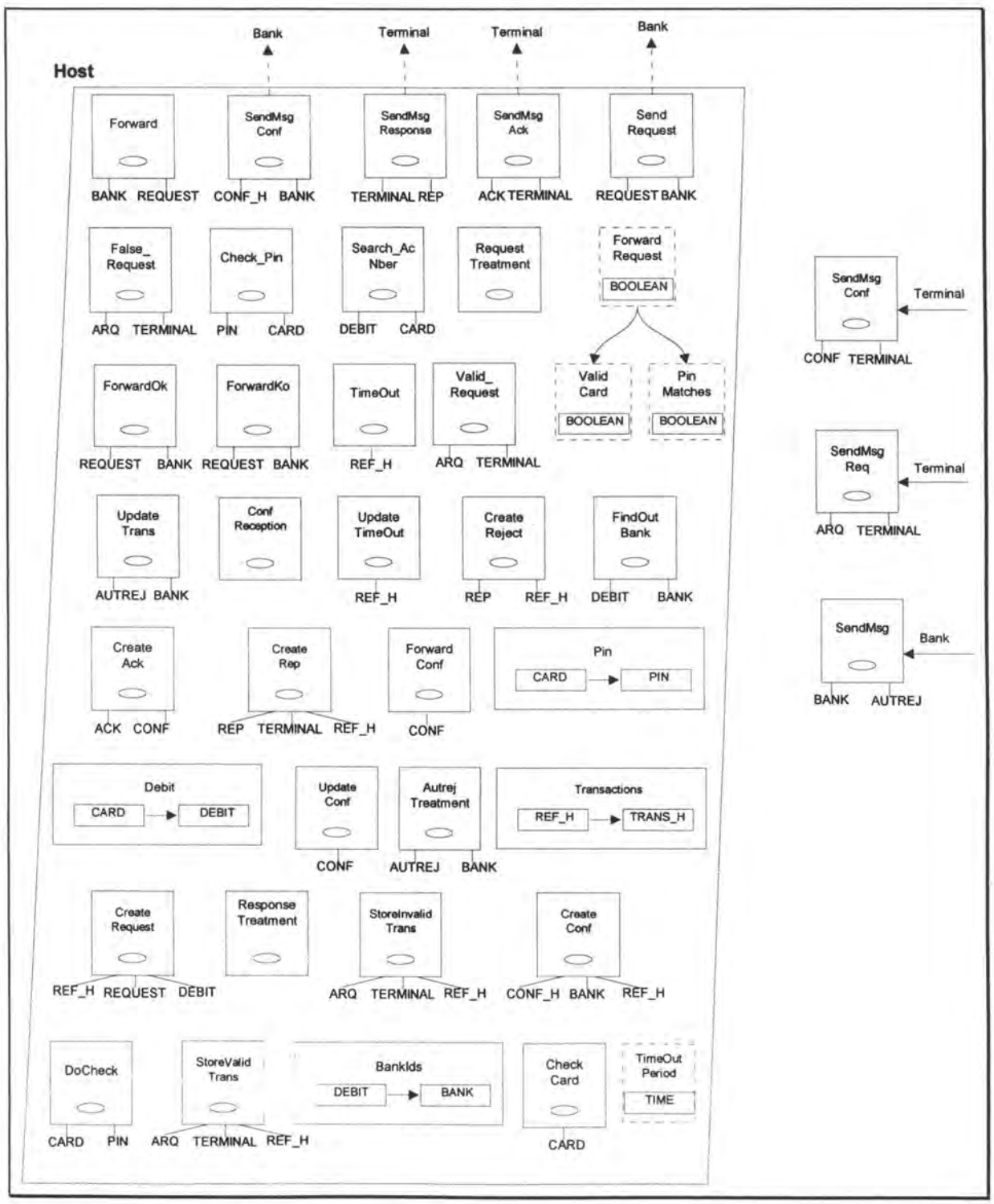


Figure 1.6. Declaration of the Host agent

Section 3. The Albert's Constraints Component

The Albert textual part allows us to express constraints that the basic components (defined in the graphical agent's declaration) have to respect and satisfy. The different constraints are expressed by using logical and temporal expressions. They are grouped under different headings; each heading representing a special characteristic of the described system.

The headings, at their turn, are split in four different sections : the Basic Constraints section , the Declarative Constraints section, the Operational Constraints section and at the Cooperation Constraints section.

The Basic Constraints section allows us to specify the initial valuation of the different state components i.e. the value of the different state components at the beginning of the agent's life-cycle as well as the derivation rules for the derived components.

The constraints of the Declarative section allow us to restrict the possible evolution of an agent by specifying the possible values that the agent's state component can take as well as the possible chains of actions. The duration of the different action can also be specified.

The Operational Constraints section describes additional characteristics of an action like for instance its triggering condition, its precondition or the effect that the action has on the agents' state components.

Let us remark that the Declarative Constraints and the Operational Constraints sections replace the Local Constraints section contained in the previous Albert version.

Finally, the Cooperation Constraints section, as its name indicates, allows us to define constraints related to the cooperation of the different agents.

We analyse now the different sections one by one. Each explained notion will be illustrated by one or several examples taken from the Four Messages Protocol (4MP) example introduced in Section 1 of this chapter.

The Basic Constraints

a1. Derived Components Constraints

A Derived Components Constraint specifies how the value of a derived component is obtained, based on the value of other state components. The wavy line in Figure 1.6. describes the existence of a special relation between three state components : the *ForwardRequest*, the *ValidCard* and *PinMatches* state components.

As we previously mentioned, the decision whether a received request can be forwarded or not to the customer's bank depends on the validity of the received debit card number and PIN code. If we assume that the validity of the received debit card number is expressed by the boolean state component *ValidCard* (*ValidCard* is true if the received debit card number is valid, false otherwise) and that the validity of the received PIN code is represented by the *PinMatches* state component (*PinMatches* is true if the received pin code is valid, false

otherwise), the Host's decision, represented by the boolean *ForwardRequest* state component, can be obtained by applying the logical *AND* operator to the two boolean state components *ValidCard* and *PinMatches*. We assume that the value of the *ValidCard* and the *PinMatches* state components are obtained by executing the *CheckCard* respectively the *CheckPin* action.

The derivation rule for the derived component *ForwardRequest* is written as :

$$\text{ForwardRequest} \triangleq \text{ValidCard} \wedge \text{PinMatches}$$

The derivation rule stipulates that the boolean state component *ForwardRequest* is true if both boolean state components *ValidCard* and *PinMatches* are true. Is the value of one of the two state components equal to false, the value of the derived component will be false too.

a2. Initial Valuation Constraints

By using the initial valuation constraints, we can assign a certain value to an internal state component at the beginning of an agent's life-cycle i.e. before any action is executed by the corresponding agent.

As mentioned earlier, the Host agent has to store the transaction information into a table in order to reduce the amount of transferred information and in order to trace back the different operations made by its customers. We assume that the transaction table is empty at the beginning of the Host's life-cycle. The corresponding Albert II constraint is given by:

$$\text{Transactions}[i] := \text{undef}$$

The Declarative Constraints

The Declarative Constraints allow us to describe a particular agent from a historical viewpoint i.e. by specifying its evolution in time. The Declarative Constraints contains three different headings : (i) the State Behaviour, (ii) the Action Composition and (iii) the Action Duration subsection.

b1. State Behaviour Constraints

The State Behaviour Constraints allow us to specify constraints that restrict the evolution of the state components of a particular agent. The constraints are expressed by using First Order Logic as well as Temporal Logic.

Real-time temporal logic operators that can be used are given by Table 1.1.. The special symbols ϕ and φ are used to represent real-time logic expressions. The r symbol represents any real-time quantity and is of rational type. Time units that may be used are seconds ("), minutes ('), hours (h), days (d) and so on.

b2. Action Composition Constraints

In Albert, an agent possesses state components and may execute a certain number of actions.

For each of those actions, the analyst may specify the action's parameters. Among those characteristics, the analyst may for instance describe the moment or condition when a particular action has to be executed.

Futr$_r$ ϕ is true at time c iff ϕ is true at time $c + r$.
Past$_r$ ϕ is true at time c iff ϕ is true at time $c - r$.
AlwF ϕ is true at time c iff ϕ is true for all t strictly greater than c.
AlwP ϕ is true at time c iff ϕ is true for all t strictly less than c.
Alw ϕ is true at time c iff ϕ is true for all t.
SomF ϕ is true at time c iff ϕ is true for at least one t strictly greater than c.
SomP ϕ is true at time c iff ϕ is true for at least one t strictly less than c.
Som ϕ is true at time c iff ϕ is true for at least one t.
Lasts$_r$ ϕ is true at time c iff ϕ is true for all t strictly between c and $c + r$.
Lasted$_r$ ϕ is true at time c iff ϕ is true for all t strictly between $c - r$ and c.
WithinF$_r$ ϕ is true at time c iff ϕ is true for at least one t strictly between c and $c + r$.
WithinP$_r$ ϕ is true at time c iff ϕ is true for at least one t strictly between $c - r$ and c.
ϕ **Until** φ is true at time c iff there is a r such that φ is true at $c + r$ and ϕ is true for all t strictly between c and $c + r$.
ϕ **Until!** φ is true at time c iff there is a r such that φ is true at $c + r$ and ϕ is true for all t strictly between c and $c + r$, and ϕ is false at $c + r$.
ϕ **Since** φ is true at time c iff there is a r such that φ is true at $c - r$ and ϕ is true for all t strictly between $c - r$ and c.
ϕ **Since!** φ is true at time c iff there is a r such that φ is true at $c - r$ and ϕ is true for all t strictly between $c - r$ and c , and ϕ is false at $c - r$.
NextTime$_r$ ϕ is true at time c iff there is a r such that ϕ is true at $c + r$ and is false for all t strictly between c and $c + r$.
LastTime$_r$ ϕ is true at time c iff there is a r such that ϕ is true at $c - r$ and is false for all t strictly between $c - r$ and c.

Table 1.1. Real-time temporal logic operators

In Albert, two different mechanism can be used in order to describe that particular moment or condition. First, an action's execution may be based on the value of a given state component.

Second, the execution of a particular action can be linked to the execution of another action. In order to specify the execution of those kind of actions, the Action Composition constraints may be used.

By using an Action Composition constraint, we can for instance specify that the Host has to execute the *DoCheck* action once the terminal has executed the *SendMsgReq* action. In our example, the Host executes the *DoCheck* action after the terminal has finished its action as the used operator is the sequential (<>) operator.

RequestTreatment ↔ Terminal.SendMsgReq(arq, term_id) < >
DoCheck(Card.arq, Pin.arq) <>
(Valid_ Request(arq, term_id) ⊕ False_ Request(arq, term_id))

Once the DoCheck action finished, the Host executes either the Valid_Request or the False_Request action as the OR operator (⊕) has been used. The Host, however, can not decide by its own which action it executes. This means that the analyst has to specify clearly the condition under which each action is executed. In our example, a constraint of the Precondition Constraints section is used to specify when which action has to be executed.

Other operators that can be used in an Albert II specification, are :

- the multiple occurrence operator $\{\}^n$ where *n* represents the number of occurrences
- the exclusive-or operator '⊕'
- the in any order operator ' | | '
- temporal operators like :
  - *action a* |⇉ *action b* meaning that action *a* and action *b* start simultaneously
  - *action a* ⇉| *action b* meaning that both actions *a* and *b* end simultaneously
  - *action a* |⇔| *action b* meaning that both actions *a* and *b* start and end simultaneously
  - the sequential operator '< *n* > ' where *n* represents the number of time units between the execution of the two actions.

In order to make the distinction between actions which may and those which may not occur outside the context of a composed action, the Action Composition Constraints section enumerates the actions which may not occur outside the context of a composed action. The enumeration is contained between the {} symbol.

b3. Action Duration Constraints

In certain cases, it is of a certain interest to specify the time needed to execute a certain action. Usual used time units are seconds ("), minutes ('), hours (h), days (d) and so on. The stated time period can be lower, equal or higher than a given time period.

In our Four Messages Protocol example, we specify that the execution of the *ForwardOk* action may not last more than the time limit represented by the *TimeOutPeriod* state component. The execution of the *ForwardKo* action, at its turn, must last more than *TimeOutPeriod* time units.

|ForwardOk(request, bank_id) |	≤ TimeOutPeriod
|ForwardKo(request, bank_id) |	> TimeOutPeriod

The Operational Constraints

In the Albert framework, an agent is represented by its internal state components, actions and cooperations. As we have seen until now, an Albert specification allow us to specify constraints that have to be satisfied or respected by a state component or by an action. The existing links between an action and a state component are described by using a constraint of the Operational Constraints Section.

The Operational Constraints Section regroups (i) the Precondition Constraints, (ii) the Effect of Action and (iii) the Triggering Constraints subsection.

c1. Precondition Constraints

As it name indicates, the Precondition Constraints allow us to express a condition that has to be satisfied before a particular action can be executed.
In our *Four Messages Protocol* example, the Host for instance has to store the received transaction information into its local database in order to reduce the number of exchanged information. The storage of the information also allows it to trace back the operations executed by its customers.

We assume that the storage of a valid transaction request at the Host's level is obtained by executing the *StoreValidTrans* action. As previously stored information have to kept save and do not have to be deleted or overwritten, the Host may only store the received information in its database at a location which does not contain information about previous stored transactions. As the different cells of the transaction table used to store the information have been initialized to *undef* (by a constraint of the Initial Valuation Constraints section), the corresponding Precondition Constraint is given by :

```
StoreValidTrans(arq, term_id, host_ref) : Transactions[host_ref] = undef
```

c2. Effects of Actions Constraints

Executing an internal or external action usually modifies the content of one or several internal states of an agent.

As the Albert language allows us to specify actions that may last a certain number of time units, the action's effect can theoretically occur at the beginning or and the end of its execution. In order to specify the moment where the effect takes place, the bracket symbol *[ ]* is used.

By convention, if the bracket *[ ]* symbol is followed by an expression, the by the expression described effect takes place at the end of the action's execution. In the case where the expression is followed by the bracket *[ ]* symbol, the effect occurs at the beginning of the action's execution.

By combining both representations, we can express the fact that several effects occur, one (or several) at the beginning and one (or several) at the end of the action's execution.

In our *Four Messages* Protocol example, the Host has to check the validity of the received PIN code and debit card number after the reception of the terminal's transaction request (ARQ) message. We assume that these checks are done by executing two actions : the *CheckCard* and the *CheckPin* action.

The first action, *CheckCard,* analyses the validity of the received debit card number *Card.arq*. The result of this check is stored in the *ValidCard* state component. The boolean state component will be true if the Pin table component contains a record indexed by the received debit card number *Card.arq*.

```
CheckCard(Card.arq) : [ ]
                        ValidCard := In(Card.arq, Pin)
```

The second action *CheckPin* analyses the validity of the received PIN code by comparing the received PIN code with the PIN code associated to the received debit card number in the Pin table. The result of the check is stored in the *PinMatches* state component.

```
CheckPin(Card.arq, Pin.arq) : [ ]
                        PinMatches := (Pin.arq = Pin[Card.arq])
```

In both cases, we assume that the effect of action takes place at the end of the corresponding action's execution.

c3. Triggering Constraints

In certain cases, it is of a certain interest to create a link between the value of a given state component and the execution of a particular action. In order to create such a link, a constraint from the Triggering Constraints section can be used. The notation used for a triggering constraint is given by :

$$<exp> / t \rightsquigarrow < action\ a >$$

representing the fact that the action *<action a>* is triggered in the case where the expression *exp* has been realized (satisfied) for *t* time units.

Until now, the different introduced Albert notions have been illustrated by examples based on the Host agent. As no Triggering constraints are used in the Host's declaration, we illustrate the Triggering notion by an example based on the Terminal agent.

In order to guarantee that the Confirmation messages successfully arrive at the Host's location, we previously assumed that the messages have to be stored in the terminal's internal memory and that they have to be sent by the terminal in regular intervals as long as no Acknowledge message is received from the Host in return. As the reception of a valid Acknowledge message implies the removal of the corresponding Confirmation message from

the terminal's memory, we can say that the Confirmation messages are sent as long as they are kept in the terminal's memory. The corresponding Triggering constraint is given by :

Memory[i] ≠ undef / SendConfFrequency ⇝ SendConfirm(i)

representing the fact that the *SendConfirm* action is executed in the case where the terminal's memory *Memory* contains the transaction information more than *SendConfFrequency* time units. The *SendConfFrequency* constant is used in order to represent the fact that the messages are sent in prefixed intervals.

The Cooperation Constraints

The different sections of the Cooperation Constraints allow the analyst to specify the existing links between an actor and its environment. Whereas the graphical Albert part only describes the importation and exportation of state components and action related information, the textual part i.e. more precisely the cooperation constraints allows us to define conditions under which those information are perceived or hidden to external agents.

Three operators (K, F, XK) are used to specify conditions under which information are perceived ('K'), hidden ('F') or exclusively perceived ('XK') by an agent. Exclusively perceived means that an agent only perceives an information or is only informing another agent if the stated condition is true. Is the stated condition not realized, the information are not perceived or communicated.

The 'simple' perception operator ('K') is not so strictly defined. An agent may perceive an information or may communicate information to another agent even if the stated condition is not realized.

Four different headings are used in the Cooperation Constraints section to distinguish between cases where an agent is communicating its internal states to other agents (state information), is informing other agents about the execution of an internal action (action information), is perceiving states from external agents (state perception) or perceives the fact that an external action has been executed (action perception) by an external agent.

The Host, for instance, always informs the bank agent when it is executing the *SendRequest* action.

XK(SendRequest(request, bank_id).Bank / true)

In return, the bank agent always communicates its execution of the *SendMsg* action to the Host. An exclusive perception is used in order to specify the fact that the Host is only informed when the bank is executing the specified action. Is this not the case, the bank has not the right to communicate the execution of the action to the Host.

XK(Bank.SendMsg(autrej, bank_id) / true)

The graphical description of the different agents of our *Four Messages Protocol* example as well as the constraints that have to be respected by the different agents are described in the following section.

Section 4. The Albert specification of the *Four Messages Protocol* example

Section 4.1. Declaration of the Data Types

BASIC TYPES

In our *Four Messages Protocol* example, different Basic Types are defined. Table 1.2 describes in the first column the name associated to each basic type and gives in the second column a short description of the type.

Name of the Basic Type	*Description*
CARD	represents the structure of the customer's debit card number
PIN	describes the structure of the PIN code associated to the customer's debit card
DATE	corresponds to the format of the transaction date
TIME	represents the format of the transaction time
DEBIT	describes the structure of the customer's bank account number
REF	represents the structure of the Terminal's reference key
REF_H	represents the structure of the Host's reference key
REF_BK	represents the structure of the Bank's reference key

Table 1. 2. Message Protocol Basic Types

CONSTRUCTED TYPES

The ARQ type describes the different sub-components of the transaction request (ARQ) message.

The state component associated to the ARQ type contains the date (Date) and time (Time) when the transaction is requested, the customer's debit card number (Card), the transaction amount (Price) and the customer's secret PIN code (Pin).

ARQ = CP (Date : DATE ; Time : TIME ; Card : CARD ; Price : INTEGER ; Pin : PIN)

The Authorization Reject (AUTREJ) message sent from the Host to the terminal is of type REP. It contains the date (Date) and time (Time) when the transaction has been requested, the customer's debit card number (Card), the transaction amount (Price), the response to the transaction request (Response), the reason (Reason) why the request has been refused (in the case where it has been refused, otherwise the field is empty) and the Host's reference key (Host_ref).

REP = CP (Date : DATE ; Time : TIME ; Card : CARD ; Price : INTEGER ;
Response : STRING ; Reason : STRING ; Host_ref : REF_H)

The CONF type, associated to the Confirmation message, sent from the terminal to the Host, contains the Host's reference key (Host_ref), the customer's confirmation (Response) and the terminal's key reference (Ref)

CONF = CP (Host_ref : REF_H ; Response : STRING ; Ref : INTEGER)

The structure of the Acknowledge message, sent from the Host to the terminal, is specified by the ACK type. An Acknowledge message contains the terminal's reference number (Ref) as well as the Host's reference key (Host_ref).

ACK = CP (Ref : INTEGER ; Host_ref : REF_H)

The TRANS_T type is used to define the structure of the information stored in the terminal's memory. The information contain the customer's confirmation response (Confirm) and the Host's reference key (Host_ref)

TRANS_T = CP (Confirm : BOOLEAN ; Host_ref : REF_H)

The structure of the Authorization request (ARQ) message, sent from the Host to the customer's bank, is specified by the REQUEST type. The sent message contains the customer's bank account number (Debit_nber), the transaction amount (Price), the Host's reference key (Host_ref) and the date (Date) and time (Time) when the transaction has been requested.

REQUEST = CP (Debit_nber : DEBIT, Price : PRICE ; Host_ref : REF_H, Date : DATE,
Time : TIME)

The information stored in the Host's transaction table are of type TRANS_H. They contain the date (Date) and time (Time) when the transaction has been requested, the customer's debit card number (Card), the transaction amount (Price), the transaction's status (Status), the reason why the transaction has been refused (in the case where the requested transaction has been refused), the identification code of the customer's bank (Bank_id), the bank's reference key (Bank_ref) and the terminal's identification code (Term_id).

TRANS_H = CP (Date : DATE ; Time : TIME ; Card : CARD ; Price : INTEGER ;
Status : STATUS_H; Reason : REASON_H, Bank_id : BANK ;
Bank_ref : REF_BK ; Term_id : TERMINAL)

The structure of the Confirmation message, sent from the Host to the customer's bank, is defined by the CONF_H type. The message contains the customer's confirmation of the transaction amount (Conf) and the bank's reference key (Bank_ref).

CONF_H = CP (Conf : STRING, Bank_ref : REF_BK)

The TRANS_BK type depicts the information that are stored at the bank's level. The stored information are the transaction date (Date) and time (Time), the customer's account number (Debit_number), the transaction amount (Price), the status of the transaction (Response), the reason of the transaction's status (Reason) and the Host's reference number (Host_ref).

TRANS_BK = CP (Date : DATE ; Time : TIME ; Debit_number : DEBIT ;
Price : INTEGER ; Response : STATUS_BK ; Reason : REASON_BK,
Host_ref : REF_H)

The information contained in the Authorization / Reject (AUTREJ) message, sent from the bank to the Host, contain the bank's response (Response), the reason of the transaction's refusal (in the case where the transaction has been refused by the bank, blank otherwise), the Host's reference key (Host_ref) and the bank's reference key (Bank_ref).

AUTREJ =: CP (Response : STRING ; Reason : STRING ; Host_ref : REF_H,
Bank_ref : REF_BK)

The value that can be associated to the transaction status at the Host's level are enumerated by the STATUS_H type.

STATUS_H = ENUM ['Transaction Requested', 'Transaction Accepted', 'Transaction Refused', 'TimeOut', 'Transaction Validated', 'Transaction Cancelled']

The value that can be assigned to the transaction status at the bank's level are enumerated by the STATUS_BK type.

STATUS_BK = ENUM['Transaction Accepted', 'Transaction Refused', 'Transaction Validated', 'Transaction Cancelled']

The transaction reject reason at the Host level is specified by the REASON_H type.

REASON_H = ENUM['', 'Invalid Pin or Card Number']

Possible values of the transaction reject reason are given by the REASON_BK type.

REASON_BK = ENUM['', 'Amount not Covered']

Section 4.2. The Terminal Agent

Declarations

STATE COMPONENTS

TimeOutPeriod instance-of TIME
The TimeOutPeriod state component specifies the number of time units the terminal waits, after the sending of the authorization request message, in order to get a response from the Host agent. If no answer arrives during that time period, a Timeout occurs at the terminal's level and the current transaction process is aborted.

SendConfFrequency instance-of TIME
The SendConfFrequency state component defines the number of time units needed by the terminal to execute the SendConfirm action. As the terminal sends the customer's transaction confirmation to the Host as long as it gets no response in return, the SendConfFrequency state component describes the frequency of the sent messages.

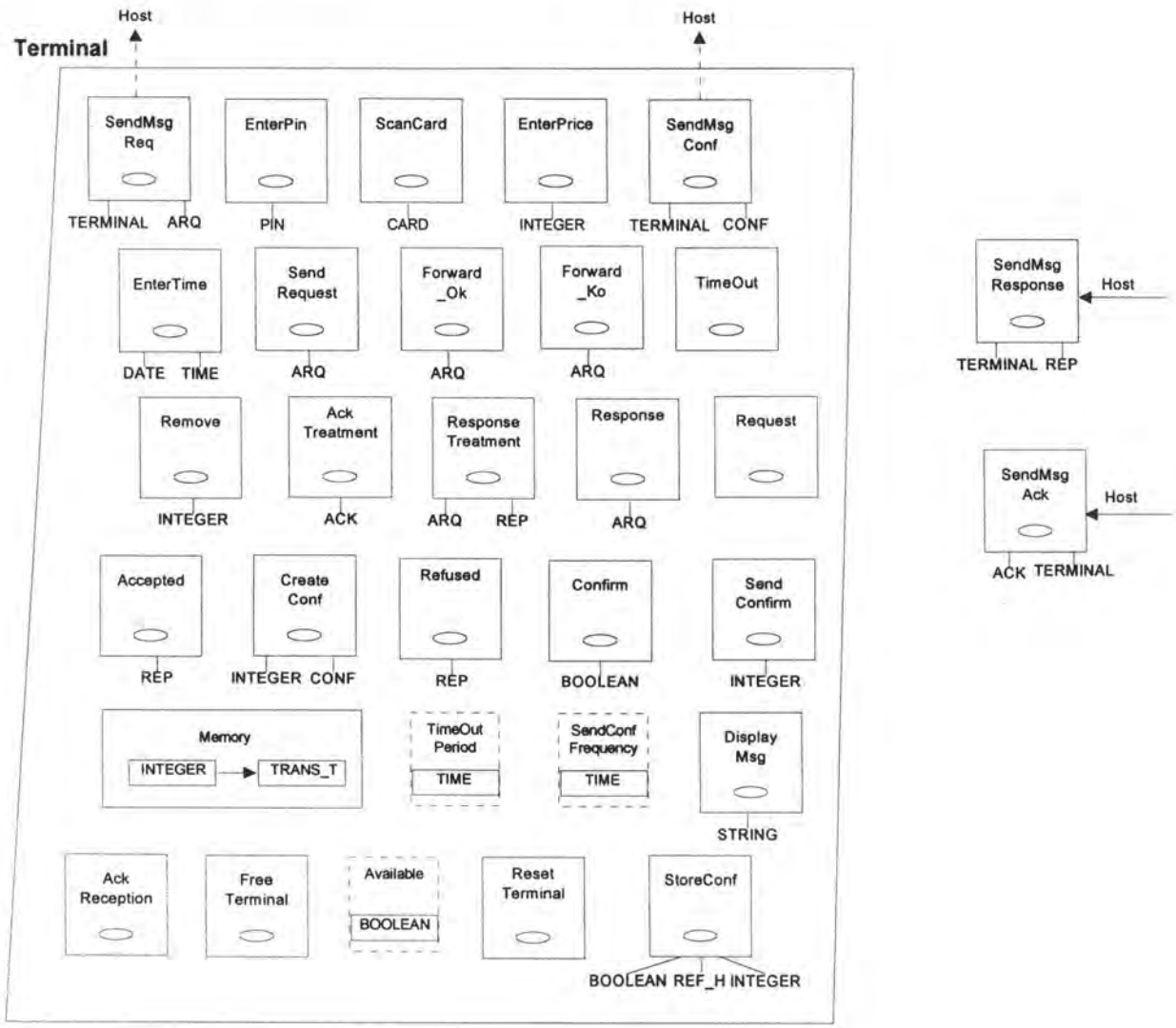
Available instance-of BOOLEAN
The Available state component is true if the terminal is available (i.e. can be used for a new transaction process), false otherwise.

Memory table-of TRANS_T indexed-by INTEGER
The Memory state component represents the terminal's internal permanent memory.

ACTIONS

ScanCard(card) : the action of scanning the customer's debit card *card* by using the terminal's card reader
ScanCard(CARD)
EnterTime(date, time) : the action of entering the transaction date *date* and time *time*
EnterTime(DATE, TIME)
EnterPrice(price) : the action of entering the transaction amount *price* by using the terminal's keyboard
EnterPrice(INTEGER)
EnterPin(pin) : the action of entering the customer's PIN Code *pin* by using the terminal's keyboard
EnterPin(PIN)
SendMsgReq(arq, term_id) : the action of sending the transaction request *arq* and the terminal's identification code *term_id* to the Host
SendMsgReq(ARQ, TERMINAL)→HOST
DisplayMsg(msg) : the action of displaying a message *msg* on the terminal's LCD display
DisplayMsg(STRING)
ResetTerminal : the action of freeing the terminal
ResetTerminal
Confirm(confirm) : the action of entering the customer's validation *confirm*
Confirm(BOOLEAN)
StoreConf(confirm, host_ref, i) : the action of storing, at position *i*, the customer's confirmation *confirm* and the Host's reference key *host_ref* in the terminal's memory
StoreConf(BOOLEAN, REF_H, INTEGER)
CreateConf(i, conf) : the action of creating a confirmation message *conf* based on the information stored at position *i* in the terminal's memory
CreateConf(INTEGER, CONF)
SendMsgConf(conf, term_id) : the action of sending the created confirmation message *conf* and the terminal's identification code *term_id* to the Host
SendMsgConf(CONF, TERMINAL)→HOST
Remove(i) : the action of removing a stored ARQ- message, located at position i, from the terminal's memory
Remove(INTEGER)

The graphical declaration of the Terminal agent



The Constraints Component of the Terminal agent

BASIC CONSTRAINTS

DERIVED COMPONENTS

INITIAL VALUATION

Memory[i] := undef
Available := true
At the beginning of the terminal's life-cycle, its internal permanent memory is empty and the terminal is available for a new transaction process.

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

In-Dom (Memory, conf) ⇒ SomeF(¬ (In-Dom (Memory, conf)))
A customer's transaction confirmation stays only a certain time in the terminal's memory.

ACTION COMPOSITION

{Host.SendMsgResponse, ScanCard, EnterTime, EnterPrice, EnterPin, SendRequest, Forward_Ok, Forward_Ko, SendMsgReq, TimeOut, DisplayMsg, FreeTerminal, ResetTerminal, ResponseTreatment, Accepted, Refused, Confirm, Host.SendMsgAck, StoreConf, SendConfirm, CreateConf, SendMsgConf, AckTreatment, Remove}

Request ↔ ScanCard(Card.arq) <> EnterTime(Date.arq, Time.arq) <>
EnterPrice(Price.arq) <> EnterPin(Pin.arq) <> SendRequest(arq)

SendRequest(arq) ↔ (Forward_Ok(arq) ⊕ Forward_Ko(arq))

Forward_Ok(arq) ↔ SendMsgReq(arq, term_id) <> Response(arq)

Forward_Ko(arq) ↔ SendMsgReq(arq, term_id) <> TimeOut

TimeOut ↔ DisplayMsg('Transaction Cancelled') <> FreeTerminal

FreeTerminal ↔ DisplayMsg('Available') <> ResetTerminal

Response(arq) ↔ Host.SendMsgResponse(rep, term_id) <> (ResponseTreatment(arq, rep)
⊕ dac)

ResponseTreatment(arq, rep) ↔ (Accepted(rep) ⊕ Refused(rep))

Accepted(rep) ↔ DisplayMsg(Price.rep) <> Confirm(confirm) <>
StoreConf(confirm, Host_ref.rep, i) <> FreeTerminal

SendConfirm(i) ↔ CreateConf(i, conf) <> SendMsgConf(conf, term_id)

Refused(rep) ↔ DisplayMsg('Transaction Refused' + reason.rep) <> FreeTerminal

AckReception ↔ Host.SendMsgAck(ack, term_id) <> (AckTreatment(ack) ⊕ dac)

AckTreatment(ack) ↔ Remove(ref.ack)

ACTION DURATION
|Forward_Ok(arq)| ≤ TimeOutPeriod
The execution of the Forward_Ok action can not last more than TimeOutPeriod time units
|Forward_Ko(arq)| > TimeOutPeriod
The execution of the Forward_Ko action must last more than TimeOutPeriod time units

OPERATIONAL CONSTRAINTS

PRECONDITION

StoreConf(_, _, i) : Memory[i] = undef

The terminal has not the right to erase previously stored information.

ResponseTreatment(arq, rep) : Date.arq = Date.rep ∧ Time.arq = Time.rep ∧
Card.arq = Card.rep) ∧ ¬Available

The ResponseTreatment action is executed in the case where the received message *rep* corresponds to the actual transaction request *arq* and the terminal is still used i.e. is not available.

AckTreatment(ack) : Host_ref.ack = Host_ref.Memory[Ref.ack]

The terminal executes the AckTreatment action in the case where the received Acknowledge *ack* message corresponds to the in the memory stored information.

Accepted(rep) : Response.rep = 'Transaction Accepted'

In order to execute the Accepted action, the transaction request must have been accepted.

Refused(rep) : Response.rep = 'Transaction Refused'

The terminal executes the Refused action in the case where the transaction has been refused.

Request : Available

A new transaction process can only be started if the terminal is available.

EFFECTS OF ACTIONS

StoreConf(confirm, Host_ref.rep, i) : []
Memory[i] := confirm, Host_ref.rep

The StoreConfirm action stores the customer's confirmation and the Host's reference key into its internal memory.

Remove(i) : []
Memory[i] := undef

The information stored at position i in the terminal's memory are removed.

ResetTerminal : []
Available := true

The terminal is reset and becomes available for a new transaction process.

ScanCard(Card.arq) : []
Available := false

The terminal becomes busy after the customer's debit card is scanned through the terminal's card reader.

TRIGGERINGS

Memory[i] ≠ undef / SendConfFrequency → SendConfirm(i)

For each stored transaction, the SendConfirm action has to be executed. We assume that the information have to stay SendConfFrequency time units in memory before the SendConfirm action is triggered.

COOPERATION CONSTRAINTS

STATE PERCEPTION

ACTION PERCEPTION

STATE INFORMATION
ACTION INFORMATION

XK(SendMsgReq (arq, term_id).Host / true)
XK(SendConfMsg (conf, term_id).Host / true)

The terminal always informs the Host when it executes the SendMsgReq respectively the SendConfMsg action. The fact that the messages do not always arrive at their destination is described by omitting the corresponding Action Perception constraint at the Host's level.

Section 4.2. The Host Agent

Declarations

STATE COMPONENTS

TimeOutPeriod instance-of TIME
The TimeOutPeriod state component specifies the number of time units the Host waits, after it has forwarded the customer's request, in order to get a response from the customer's bank. If no valid answer arrives during that time period, a TimeOut occurs at the terminal's level and the current transaction process is aborted.

ValidCard instance-of BOOLEAN
The ValidCard state component is true, if the Host has received a valid debit card number.

PinMatches instance-of BOOLEAN
The PinMatches state component is true, if a valid Pin Code has been entered.

ForwardRequest instance-of BOOLEAN
The received transaction request is forwarded to the customer's bank if the value of the ForwardRequest state component is true. Otherwise, the terminal is informed that the requested transaction has been refused. The ForwardRequest state component is a derived component and its value is derived from the ValidCard and PinMatches state components.

Pin table-of PIN indexed-by CARD
The Pin table contains all the debit card numbers emitted by the Host as well as their corresponding PIN (Personnal Identification Number) codes.

Debit table-of DEBIT indexed-by CARD
The Debit table contains for each debit card number the corresponding account number.

Transactions table-of TRANS_H indexed-by REF_H
The Transactions table is used by the Host agent in order to store the different transaction information.

BankIds table-of BANK indexed-by DEBIT
The BankIds table contains for each debit card number the corresponding identification code of the bank.

ACTIONS

CheckPin(card, pin) : the action of checking if the entered Pin Code *pin* corresponds to the received card number *card*

CheckPin(CARD, PIN)

Search_AcNber(card, debit_nber) : the action of searching the account number *debit_nber* corresponding to the received card number *card*

Search_AcNber(CARD, DEBIT)

StoreValidTrans(arq, term_id, host_ref) : the action of storing a valid transaction request *arq* as well as the terminal's identification code *term_id* at position *host_ref* into the transactions table

StoreValidTrans(ARQ, TERMINAL, REF_H)

CreateRequest(debit_nber, host_ref, request) : the action of creating a request message *request* based on the customer's account number *debit_nber* as well as different information previously stored in the transactions table at position *host_ref*

CreateRequest(DEBIT, REF_H, REQUEST)

FindOutBank(debit_nber, bank_id) : the action of finding out the identification code *bank_id* of the bank to which the request message will be forwarded.

FindOutBank(DEBIT, BANK)

SendRequest(request, bank_id) : the action of forwarding the customer's transformed transaction request *request* to the customer's bank identified by *bank_id*

SendRequest(REQUEST, BANK) → BANK

UpdateTimeOut(host_ref) : the action of recording the occurrence of a Timeout by updating the transaction information of the transaction located at position *host_ref* in the transaction table.

UpdateTimeOut(REF_H)

StoreInvalidTrans(arq, term_id, host_ref) : the action of recording an invalid received transaction request *arq* as well as the terminal's identification code *term_id* into the transactions table at position *host_ref*

StoreInvalidTrans(ARQ, TERMINAL, REF_H)

CreateReject(host_ref, rep) : the action of creating a negative response message *rep* based on an invalid received transaction request *arq*

CreateReject(REF_H, REP)

SendMsgResponse(rep, term_id) : the action of forwarding the response *rep* to the terminal identified by *term_id*

SendMsgResponse(REP, TERMINAL) →TERMINAL

UpdateTrans(autrej, bank_id) : the action of updating the transactions table after the reception of the bank's response *autrej*

UpdateTrans(AUTREJj, BANK)

CreateRep(host_ref, rep, term_id) : the action of creating a response message *rep* containing the bank's decision whether the transaction request has been accepted or not. The needed information are contained in the transaction table. The identification code of the destination terminal is also determined

CreateRep(REF_H, REP, TERMINAL)

CreateAck(conf, ack) : the action of creating an acknowledge message *ack* based on the received confirmation message *conf*

CreateAck(CONF, ACK)

SendMsgAck(ack, term_id) : the action of forwarding the acknowledge message *ack* to the terminal identified by *term_id*

SendMsgAck(ACK, TERMINAL) → TERMINAL

UpdateConf(conf) : the action of updating the transaction record based on the received confirmation message *conf*

UpdateConf(CONF)

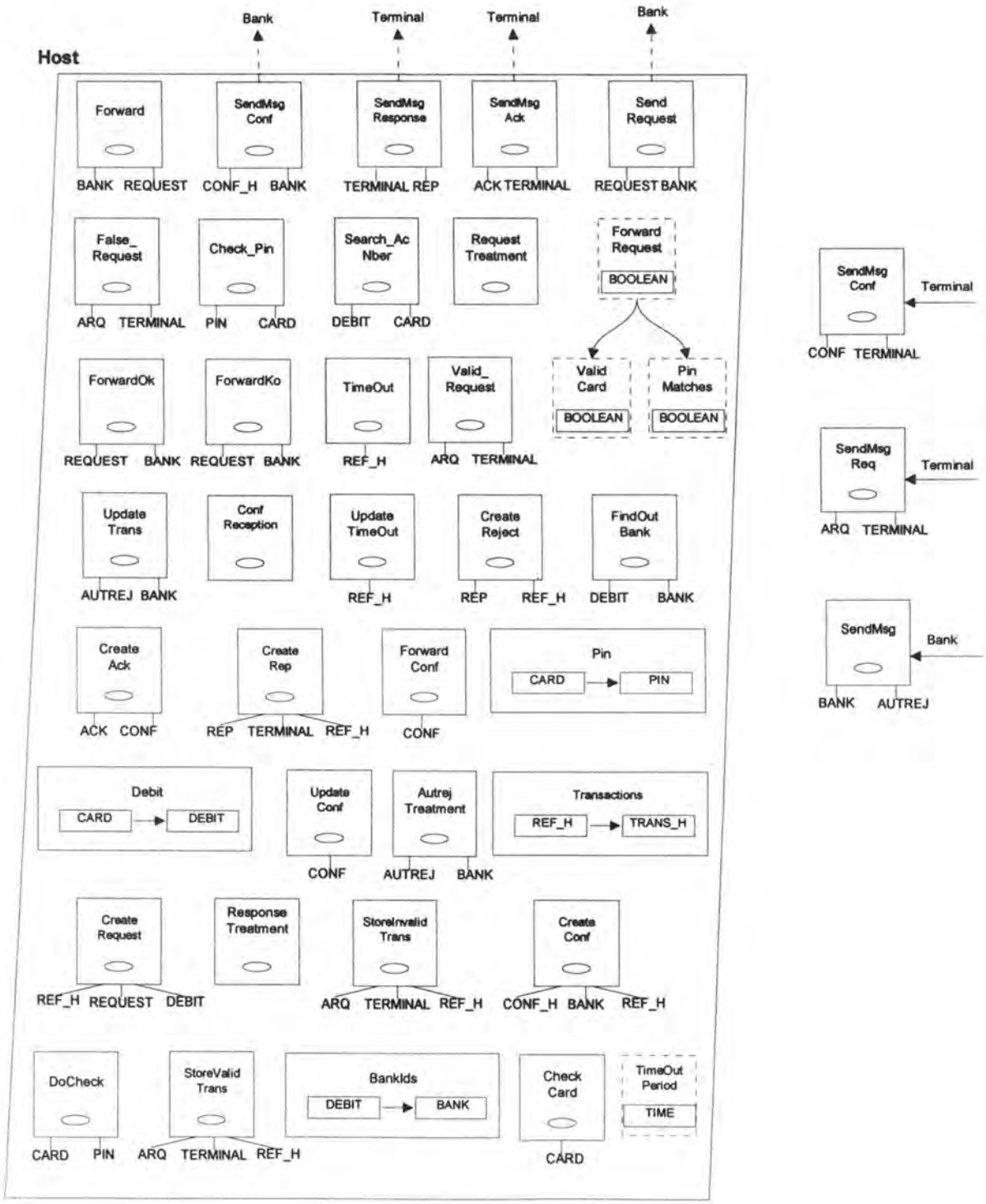
CreateConf(host_ref, conf_h, bank_id) : the action of creating a confirmation message *conf_h* and as well as the determination of the identification code of the customer's bank *bank_id* based on the information located at position *host_ref* in the transaction table.

CreateConf(REF_H, CONF_H, BANK)

SendMsgConf(conf_h, bank_id) : the action of forwarding the confirmation message *conf_h* to the customer's bank identified by *bank_id*

SendMsgConf(CONF_H, BANK) → BANK

The graphical declaration of the Host agent



The Constraints Component of the Host agent

BASIC CONSTRAINTS
DERIVED COMPONENTS

ForwardRequest ≙ ValidCard ∧ PinMatches

INITIAL VALUATION
Transactions[i] := undef
At the beginning of the Host's life-cycle, the Transactions table is empty.

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

ACTION COMPOSITION

{Terminal.SendMsgReq, Bank.SendMsg, Terminal.SendMsgConf, DoCheck,
Valid_ Request, False_ Request, CheckCard, CheckPin, Search_AcNber,
StoreValidTrans, CreateRequest, FindOutBank, Forward, ForwardOk,
ForwardKo, SendRequest, TimeOut, UpdateTimeOut, StoreInvalidTrans
CreateReject, SendMsgResponse, AutrejTreatment, UpdateTrans, CreateRep
CreateAck, SendMsgAck, ForwardConf, UpdateConf, CreateConf, SendMsgConf}

RequestTreatment ↔ Terminal.SendMsgReq(arq, term_id) <>
DoCheck(Card.arq, Pin.arq) <>
(Valid_ Request(arq, term_id) ⊕
False_ Request(arq, term_id))

DoCheck(Card.arq, Pin.arq) ↔ CheckCard(Card.arq) <> CheckPin(Card.arq, Pin.arq)

Valid_ Request(arq, term_id) ↔ Search_AcNber(Card.arq, debit_nber) <>
StoreValidTrans(arq, term_id, host_ref) <>
CreateRequest(debit_nber, host_ref, request) <>
FindOutBank(debit_nber, bank_id) <>
Forward(request, bank_id)

Forward(request, bank_id) ↔ (ForwardOk(request, bank_id) ⊕
ForwardKo(request, bank_id))

ForwardOk(request, bank_id) ↔ SendRequest(request, bank_id) <> ResponseTreatment

ForwardKo(request, bank_id) ↔ SendRequest(request, bank_id) <>
TimeOut(Host_ref.request)

TimeOut(Host_ref.request) ↔ UpdateTimeOut(Host_ref.request)

False_ Request(arq, term_id) ↔ StoreInvalidTrans(arq, term_id, host_ref) <>
CreateReject(host_ref, rep) <>
SendMsgResponse(rep, term_id)

ResponseTreatment ↔ Bank.SendMsg(autrej, bank_id) <>
(AutrejTreatment(autrej, bank_id)
⊕ dac)

AutrejTreatment(autrej, bank_id) ↔ UpdateTrans(autrej, bank_id) ◇
CreateRep(Host_ref.autrej, rep, term_id) ◇
SendMsgResponse(rep, term_id)

ConfReception ↔ Terminal.SendMsgConf(conf, term_id) ◇ CreateAck(conf, ack) ◇
SendMsgAck(ack, term_id) ◇ (ForwardConf(conf) ⊕ dac)

ForwardConf(conf) ↔ UpdateConf(conf) ◇
CreateConf(Host_ref.conf, conf_h, bank_id) ◇
SendMsgConf(conf_h, bank_id)

ACTION DURATION

|ForwardOk(request, bank_id)| ≤ TimeOutPeriod
|ForwardKo(request, bank_id)| > TimeOutPeriod

The ForwardOk action may not last more than TimeOutPeriod time units. The ForwardKo action, at its turn, may not last less than TimeOutPeriod time units.

OPERATIONAL CONSTRAINTS

PRECONDITIONS

Valid_Request(arq, term_id) : ForwardRequest
The Valid_Request action can not be executed if the Host has not received a valid transaction request.

False_Request(arq, term_id) : ¬ ForwardRequest
The False_Request action can not be executed if the Host has received a valid transaction request

StoreValidTrans(arq, term_id, host_ref) : Transactions[host_ref] = undef
The StoreValidTrans action can only store transaction information into its Transactions table at position host_ref, if the Transactions table does not contain at that location information about other previously stored transactions.

StoreInvalidTrans(arq, term_id, host_ref) : Transactions[host_ref] = undef
Information of previously stored transactions can not be overwritten.

ForwardConf(conf) : status.Transactions[Host_ref.conf] = 'Transaction Accepted'
The ForwardConf action can not be executed if the confirmation message has already been previously forwarded

AutrejTreatment(autrej, bank_id) : status.Transactions[Host_ref.autrej] ≠ 'TimeOut'
The AutrejTreatment for a given transaction can not be executed if a TimeOut has occurred for that particular transaction.

EFFECTS OF ACTIONS

CheckCard(Card.arq) : []
ValidCard := In(Card.arq, Pin)
The ValidCard state component is true, if the Host has received a valid debit card number

CheckPin(Card.arq, Pin.arq) : []
PinMatches := (Pin.arq = Pin[Card.arq])
The PinMatches state component is true, if the customer has entered a valid Pin code

StoreValidTrans(arq, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq,
Date.arq, Time.arq,
'Transaction Requested',
term_id
The Host stores information about a transaction request (that will be transferred to the corresponding bank) into its transaction table.

StoreInvalidTrans(arq, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq,
Date.arq, Time.arq,
'Transaction Refused',
'Invalid Pin or Card
Number', term_id
The Host stores information about a refused transaction request into its transaction table. The transaction request has been refused because the Host has received an invalid debit card number or Pin code.

UpdateTimeOut(Host_ref.request) : []
Status.Transactions[Host_ref.request] := 'TimeOut'
The status of a given transaction is updated due to the occurrence of a Timeout

UpdateTrans(autrej, bank_id) : []
Status.Transactions[Host_ref.autrej] := Response.autrej
Reason.Transactions[Host_ref.autrej] := Reason.autrej
Bank_id.Transactions[Host_ref.autrej] := Bank_id
Bank_ref.Transactions[Host_ref.autrej] := Bank_ref.autrej
The information of a given transaction are updated after the reception of the bank's authorization / reject message

UpdateConf(conf) : []
Status.Transactions[Host_ref.conf] := Response.conf
The status of a given transaction is updated after the reception of the terminal's Confirmation message.

TRIGGERINGS

COOPERATION CONSTRAINTS

ACTION PERCEPTION

XK(Bank.SendMsg(autrej, bank_id) / true)
The Host agent perceives the execution of the SendMsg action each time the action is executed by the bank agent.

STATE PERCEPTION

ACTION INFORMATION

XK(SendRequest(request, bank_id).Bank / true)
XK(SendMsgResponse(rep, term_id).Terminal / true)
XK(SendMsgAck(ack, term_id) .Terminal / true)
XK(SendMsgConf(conf_h, bank_id).Bank / true)
The Host agent always informs the destination agents when it is sending them a message.

STATE INFORMATION

Section 4.3. The Bank Agent

Declarations

STATE COMPONENTS

BalanceOk instance-of BOOLEAN
The BalanceOk state component is true if the customer's account balance covers the requested transaction amount.

Transactions table-of TRANS_BK indexed-by REF_BK
The Transactions table is used by the bank agent to store all the information about its customer's transactions.

Accounts table-of INTEGER indexed-by DEBIT
The Accounts table contains for each account its corresponding balance.

ACTIONS

Check_Balance(debit_card, price) : the action of checking the balance of the customer's bank account *debit*_card by taking into account the transaction amount *price*.
Check_Balance(DEBIT, INTEGER)
StoreAccepted(request, bank_ref) : the action of storing the accepted transaction request *request* into the transaction table at position *bank_ref*
StoreAccepted(REQUEST, REF_BK)
CreateRep(bank_ref, autrej) : the action of creating a response message *autrej* containing the bank's response based on the information contained in the transaction table at position *bank_ref*
CreateRep(REF_BK, AUTREJ)
SendMsg(autrej, bank_id) : the action of sending the response message *autrej* and the bank's

identification code *bank_id* to the Host agent

SendMsg(AUTREJ, BANK)→HOST

StoreRefused(request, bank_ref) : the action of storing the refused transaction request *request* into the transaction table at position *bank_ref*

StoreRefused(REQUEST, REF_BK)

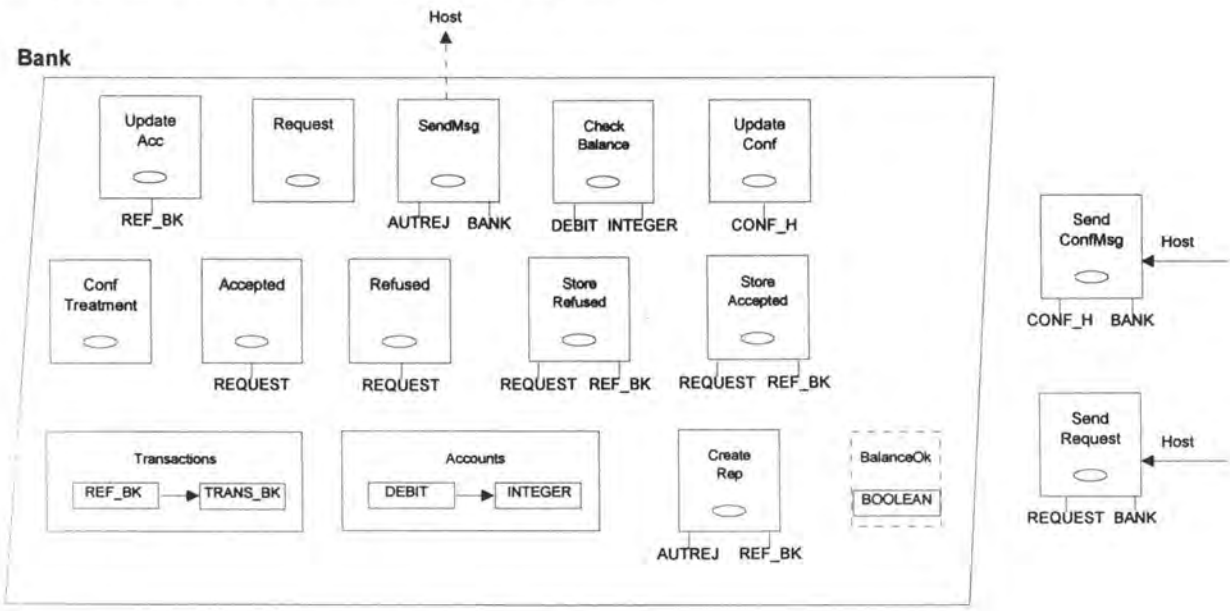
UpdateAcc(bank_ref) : the action of retrieving the transaction amount from the customer's bank account by using the information stored in the transaction table at position *bank_ref*

UpdateAcc(REF_BK)

UpdateConf(conf_h) : the action of updating the transaction record after the reception of the customer's confirmation message *conf_h*

UpdateConf(CONF_H)

The graphical declaration of the Bank agent



The Constraints Component of the Bank agent

BASIC CONSTRAINTS

DERIVED COMPONENTS

INITIAL VALUATION

Transactions[i] := undef

At the beginning of the bank's life-cycle, the Transactions table used to store the transactions made by its customers is empty

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

ACTION COMPOSITION

{Host.SendRequest, Check_Balance, Accepted, Refused, StoreAccepted, CreateRep, SendMsg, StoreRefused, Host.SendConfMsg, UpdateAcc, UpdateConf}

Request ↔ Host.SendRequest(request, bank_id) <> Check_Balance(Debit_card.request, Price.request) <> (Accepted(request) ⊕ Refused(request))

Accepted(request) ↔ StoreAccepted(request, bank_ref) <> CreateRep(bank_ref, autrej) <> SendMsg(autrej, bank_id)

Refused (request) ↔ StoreRefused(request, bank_ref) <> CreateRep(bank_ref, autrej) <> SendMsg(autrej, bank_id)

ConfTreatment ↔ Host.SendConfMsg(conf_h, bank_id) <> (UpdateAcc(Bank_ref.conf_h) ⊕ dac) <> UpdateConf(conf_h)

ACTION DURATION

OPERATIONAL CONSTRAINTS

PRECONDITION

Accepted(request) : BalanceOk

The bank has not the right to execute the Accepted action in the case where the received transaction request has not been accepted i.e. in the case where the BalanceOk state component is false.

Refused(request) : ¬ BalanceOk)

The bank has not the right to execute the Refused action in the case where the received transaction request has been accepted i.e. in the case where the BalanceOk state component is true.

UpdateAcc(Bank _ref.conf_h) / status.Transactions[Bank_ref.conf_h] = 'Transaction Accepted')

The bank has not the right to retrieve the transaction amount from the customer's account in the case where the customer has refused the at the terminal's screen displayed transaction amount.

EFFECTS OF ACTIONS

Check_Balance(Debit_nber.request, Price.request) : []
Balance_Ok :=
(Accounts[Debit_nber.request] -
Price.request) ≥0

The Balance_Ok state component is true if the transaction amount is covered by the customer's account balance.

StoreAccepted(request, bank_ref) : []
Transactions[bank_ref] := Debit_nber.request,
Price.request,

Date.request, Time.request,
'Transaction Accepted',
Host_ref.request,

The StoreAccepted action stores the information of an accepted transaction request into the Transactions table.

StoreRefused(request, bank_ref) : []
Transactions[bank_ref] := Debit_nber.request,
Price.request,
Date.request, Time.request,
'Transaction Refused',
'Amount not covered',
Host_ref.request

The StoreRefused action stores the information of a refused transaction request into the Transactions table.

UpdateAcc(bank_ref.conf_h) : []
Accounts[Debit_nber.Transactions[Bank_ref.conf_h] :=
Accounts[Debit_nber.Transactions[Bank_ref.conf_h] -
Price.Transactions[Bank_ref.conf_h]

The UpdateAcc action updates the customer's account balance once the customer's transaction validation has arrived at the bank's location.

UpdateConf(conf_h) : []
response.Transactions[Bank_ref.conf_h] := Conf.conf_h

The UpdateConf action updates the status of the corresponding stored transaction.

TRIGGERINGS

COOPERATION CONSTRAINTS

ACTION PERCEPTION

XK(Host.SendConfMsg(conf_h, bank_id) / bank_id = self)
XK(Host.SendRequest(request, bank_id) / bank_id = self)

The bank agent perceives the Host's execution of the SendConfMsg respectively the SendRequest action in the case where the sent messages are addressed to the bank agent. A message is addressed to a certain bank, if the specified identification code matches with the bank's identification code i.e. in the case where the condition *bank_id = self* is true.

STATE PERCEPTION

ACTION INFORMATION

XK(SendMsg(autrej, bank_id).Host / true)
The bank agent always informs the Host agent when it is executing the SendMsg action.

STATE INFORMATION

Chapter 2 : Introduction to the i* framework

The i* framework (pronounced i star) has been developed at the University of Toronto and has been successfully applied in Requirements Engineering, in Business Process Reengineering, in Organizational Impacts Analysis and in Software Process Modelling areas. The here below presented introduction to the i* framework is based on [5], the latest version of the i* framework.

The i* framework is composed of two components : (i) the Strategic Dependency model which allows us to represent the existing dependencies between actors of a given organization or process and (ii) the Strategic Rationale model which allows us to describe the internal behavior of the different actors specified in the Strategic Dependency model.

The structure of this chapter is similar to the previous chapter. Section 1 describes the example which is used through this chapter in order to explain the different i* framework concepts. Section 2 analyzes the first model of the i* framework : the Strategic Dependency model. Section 3, at its turn, describes the Strategic Rationale model.

Section 1. Description of the Credit Card Purchase example

A customer applies to the shopkeeper in order to buy a certain number of goods or services. We assume that the customer wants to pay the goods or services by using its credit card. Before the customer can however be regarded as the regular owner, the shopkeeper has to ensure that the transaction is covered by the Credit Card Company i.e. that he or she will be refunded by the customer's Credit Card Company.

To do this, the shopkeeper has to follow a procedure similar to the procedure described in the *Four Messages Protocol* example of Chapter 1. First, the shopkeeper has to enter the transaction amount into a terminal located in the shop and scan the customer's credit card through the card-reader of the terminal. The transaction information are then forwarded to the Credit Card Company. The Credit Card Company evaluates the received request and sends its response back to the terminal.

In the case where the transaction has been accepted, a coupon is printed out containing the main transaction information like the transaction date and time, the transaction amount and the number of the used credit card. In order to complete the transaction process, the customer has to validate the transaction by signing the printed transaction coupon.

Depending on the Credit Card Company, the shopkeeper has to compare the customer's signature on the coupon with the signature on the credit card. The Credit Card Company can also request that the shopkeeper controls the customer's identity (in order to find out if the transaction is made by the owner of the card) . In our example, both checks have to be executed.

The response of the Credit Card Company depends on three criteria : (i) the validity of the received credit card number, (ii) the actual debt level of the customer's account and (iii) the customer's Transaction History.

The customer's maximum debt level describes a level that the customer may not exceed. The Transaction History defines a limit of expenses for a given period that the customer has to respect. This limit may be different than the customer's maximum debt level.

In practice, both limits i.e. the maximum amount of debt and the Transaction History limit can vary from one company to another. In our simplified *Credit Card Purchase* example, we assume that the two limits exist but we do not specify their amount.

Section 1. The Strategic Dependency Model

Definition :

"A Strategic Dependency Model can be described as a set of nodes and arcs where the nodes represent the different actors of an organization or processes and the arcs the existing dependencies between the different actors (nodes)."

A dependency exists in a situation where an actor, called depender, depends on another actor, called the dependee, to get something realized. The object of a dependency, called dependum, can be a task to execute, a goal to achieve or a resource to furnish. Depending on the type of the dependum, the dependency arc is called a task dependency, a goal dependency, a softgoal dependency or a resource dependency.

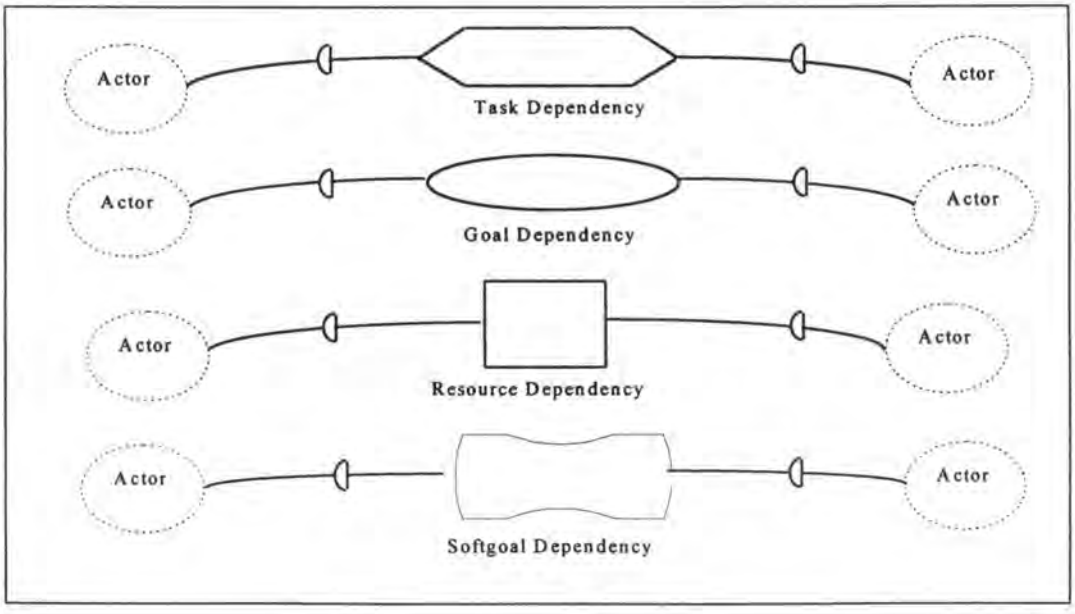


Figure 2.1. Graphical representation of the different dependency types

The graphical representation of the different types of dependency is given by Figure 2.1.

The Strategic Dependency model of our *Credit Card Purchase* example is given by Figure 2.2..

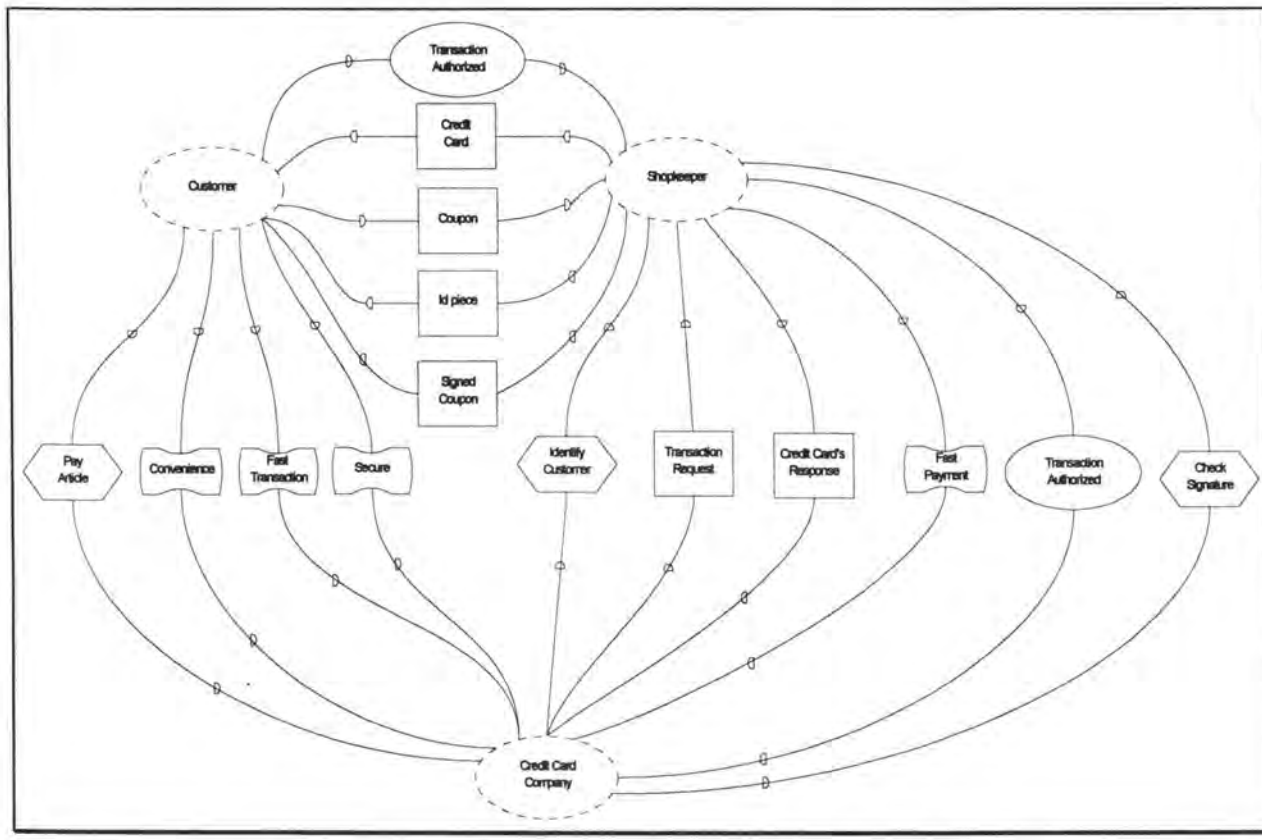


Figure 2.2. Strategic Dependency of the Credit Card Purchase example

Figure 2.2. depicts that the customer depends on the Credit Card Company in order to pay for the bought goods or services. The dependency is represented by a task dependency called *Pay Article*. The *Transaction Authorization* goal dependency between the customer and the shopkeeper expresses the fact that the customer depends on the shopkeeper in order to get the Credit Card Companies' transaction authorization. The authorization, in our example, is represented by the *Coupon* resource dependency. We also assume that the customer has some desires and wants and represent them by the *Convenience*, *Fast Transaction* and *Secure* softgoals. We analyze those softgoals later on in the softgoal dependency subsection.

The Shopkeeper depends on the customer in order to get the customer's identification piece, its credit card and the signed coupon (in the case where the requested transaction has been authorized by the customer's Credit Card Company). The different items are represented by a resource dependency. The shopkeeper depends on the Credit Card Company in order to get the transaction's authorization (represented by the *Transaction Authorization* goal dependency*)*. The authorization itself is represented by the *Credit Card's Response* resource dependency. As we assume that, in our example, the shopkeeper is refunded by the Credit Card Company, the shopkeeper's desire of getting paid as possible, is represented by the *Fast Payment* softgoal.

The Credit Card Company depends on the transaction information represented by the *Transaction Request* resource. It also depends on the shopkeeper in order to get the customer

identified (represented by the *Identify Customer* task dependency) and in order to get the customer's signature checked (depicted by the *Check Signature* task dependency).

1. Goal Dependency

Definition :

"A goal dependency describes a state or condition that has to be achieved by the dependee. The depender depends on the dependee to get a certain state or condition brought about but does not specify the way how the dependum has to be achieved".

A certain degree of liberty can be identified in a goal dependency. As the depender does not care about the way how the dependee organizes itself to bring about the dependum, the dependee can analyze different existing ways which allow the achievement of the dependum and chose the more convenient one.

As the depender is only interested in the achievement of a state or condition, the dependee could theoretically delegate its responsibilities to another actor. This new actor would then be in charge of achieving the requested dependum.

In our Credit Card Purchase example, the customer depends on the Host in order to get the transaction authorization. We assume that the customer is not interested in the way the Host has to act in order to get and transfer this authorization and represent the dependency by a goal dependency. The same remark can be made for the shopkeeper's *Transaction Authorization* goal dependency.

2. Task Dependency

Definition

A task dependency describes a situation in which the dependee depends on the depender to get a certain task executed. A particularity of the task dependency consists in the fact that the task is accompanied by a description of how to perform the task. The liberty factor of the goal dependency does no more exist. The different actors have to execute the task as requested by the description.

The Credit Card Company, for instance, depends on the shopkeeper to get the task '*Identify Customer*' and '*Check Signature*' executed. The notion of task dependency requires that both tasks have to be followed by a description on how they have to be executed.
In our example, we assume that the verification of the customer's identification is done by verifying the customer's identification piece (in order to find out if the customer is the owner of the credit card). The *'CheckSignature'* task prescribes that the shopkeeper has to compare the signature on the coupon with the signature on the customer's credit card.

3. Softgoal Dependency

Definition

In a softgoal dependency, the depender depends on the dependee to get a certain condition achieved. In opposition to a goal dependency, the depender does not specify precisely (in a sharpened way) what it expects from the dependee. A softgoal dependency has no real sense if it is not associated to a task -, goal - or resource dependency.

In our Credit Card Purchase example, the '*Pay Article*' task dependency describes the fact that the customer depends on the Credit Card Company to pay for the goods or services. The customer may however have some special wants or let us rather say some additional desires. The customer may, for instance, not feel like waiting too long in the shop before the Credit Card Company's response arrives. This fact is represented by the '*Fast Transaction*' softgoal.

The '*Fast Transaction*' softgoal provides some crucial information to the Credit Card Company especially in cases where a given system has to be improved as it reflects the customer's desires and wants. How long a customer wants to wait i.e. how fast a transaction response has to arrive is not sharply defined and may be subject to interpretations. Does the customer accept a response time of 15 seconds ? What about a delay of 30 seconds ? Is the delay more easier accepted by the customer if it is informed about the reason of the delay ?

Similar questions can be asked for the customer's '*Secure*' and '*Convenience*' softgoal. What does a customer mean by a secure transaction ? Is secure referring to a mechanism that guarantees the customer that only transactions that are executed by him are retrieved from its account or is the notion of security related to the protection of the information during the transfer between the shop and the Credit Card Company? What does a convenient way of acting mean for the customer ? How many manipulations or operations does the customer accept ? All those questions are related to the actor's softgoaols and may be subject to interpretations.

4. Resource Dependency

Definition

"A resource dependency describes a relationship between two actors in which the depender depends on the dependee in order to get a particular resource. The dependum can be a physical or an informational entity. The resource is usually reused by the depender in order to achieve a certain state or condition that is requested by another actor".

The shopkeeper, for instance, depends on the customer's credit card and identification piece in order to execute the different checks. The Credit Card Company depends on the different information included in the transaction request message so that it can determinate if the customer's transaction request can be accepted or not.

At the i* level, a resource dependency represents more than a simple non intentional flow. Depending on the background of the resource dependency, the resource dependency may be accompanied by a task, a goal or a softgoal dependency.

In our example, the customer hands over its credit card and identification piece to the shopkeeper in order to pay the goods or services i.e. in order to get the transaction authorization from the Credit Card Company. If we assume that the customer is not interested in how the authorization is made and transferred to the shop's location, the Credit Card and Id Piece resource dependencies are accompanied by a goal dependency called '*Transaction Authorization*' dependency between the customer and the Host and a task dependency called '*Pay Article*' between the customer and the Credit Card Company.

In order to close the transaction process, the shopkeeper depends on the Credit Card Companies' response. As shopkeepers usually try to maximize their benefit, a fast payment procedure is usually required so that the Credit Card Response resource dependency is accompanied by a softgoal dependency.

3. Analysis of the Strategic Dependency Model

The i* framework allows us to represent an actor as an intentional and strategic entity. An actor does not simply execute some tasks but has motivations and interests. One of its major interests consists in a long-term relationship with the other agents in order to stabilize and extent its position inside its organization or process.

The opportunities it is faced to as well as its and the others' vulnerabilities are usually analyzed by each actor. An existing dependency does not mean that a depender is executing a certain task, is achieving a certain goal or is producing a certain resource. The dependee may not be interested in or does not have the time to achieve, execute or produce the dependum.

The Strategic Dependency model allows the analyst to plan or modify an existing organization or process in a way that all the actors can be satisfied. Planning or modifying does however not mean reducing the dependencies as much as possible. Such a reduction would inevitably bring about conflicts inside the organization or process. Dependencies will always exist in an organization or process composed of autonomous components. Instead, plan and modify means that the dependencies should be equally distributed among the different actors.

In order to get an equal distributed Strategic Dependency model, certain analyses can be made. The opportunities and vulnerabilities of each actor as well as their role and position inside their organization or process have to be analyzed. The different analyzes are described in the next two subsections.

Opportunities and vulnerabilities of an actor

An opportunity for an actor consists in delivering a dependum to an actor who requires the dependum in order to achieve a certain state or condition. The delivery of the dependum represents an opportunity for the dependee as it can ask something in exchange. This can be a task, a resource or the achievement of a condition. We say that the depender is vulnerable since it depends on the dependee to get a certain task executed or a condition achieved. The

degree of vulnerability depends on the number of dependees that are able of providing the same kind of dependum. A low number of dependees implies that the depender is highly vulnerable and that the power of the dependee is very high as the dependee is one of only a few actors which are able to provide the dependum.

The Credit Card Company, for instance, depends on the shopkeeper to get the customer identified and the customer's signature checked. Both verifications represent a critical operation for the company as the company depends on the validity of the received information in order to execute its tasks. False information (for instance the non detection of the utilization of a stolen credit card) can have some serious effects on the Credit Card Company. If the utilization of the stolen card is not found out by the shopkeeper, the customer can request the canceling of the transaction. In this case, the customer is refunded and the Credit Card Company has to pay for the items.

The high vulnerability of the Credit Card Company is however compensated by a reciprocal vulnerability of the shopkeeper. First of all, we assumed that the shopkeeper is paid by the Credit Card Company. In practice, the shopkeeper is only paid if it has followed the procedures prescribed by the Credit Card Company. As it is difficult for the company to find out whether the shopkeeper has respected the prescribed procedures or not, the shopkeeper is usually forced to share the Credit Card Comapnies' costs due to the canceling of the transaction by the customer.

As shopkeepers usually try to maximize their profit, a certain vulnerability exist on both sides so that both actors can almost be assured that the requested condition or state is achieved.

Agent, Role and Position

The i* framework allows us to represent a given organization or process by taking an actor orientated approach. The Strategic Dependency model gives us information about the different actors and the dependencies existing between them.

By introducing the notion of agent, role and position, the i* framework allows the analyst to represent an actor in more details by describing whether a given dependency is linked to a particular agent or to the position it occupies in the organization or process.

In the case where a given dependency is linked to a particular agent, the agent possesses a certain knowledge that becomes crucial for the organization. The departure of that agent or its unavailability can, for instance, pose a certain problem for its organization or process.

In the case where the dependency is linked to a certain position of a given organization or process, the agent occupying that particular position can be replaced as it is the agent's position and not its knowledge which is crucial for the organization.

Based on the description of our Credit Card Company actor, the analyst can, for instance, represent the fact that the Credit Card Company is composed of several agents occupying different positions like for instance clerk agents and managment agents. The analyst can further represent the fact that certain roles are attached or played by a given position.

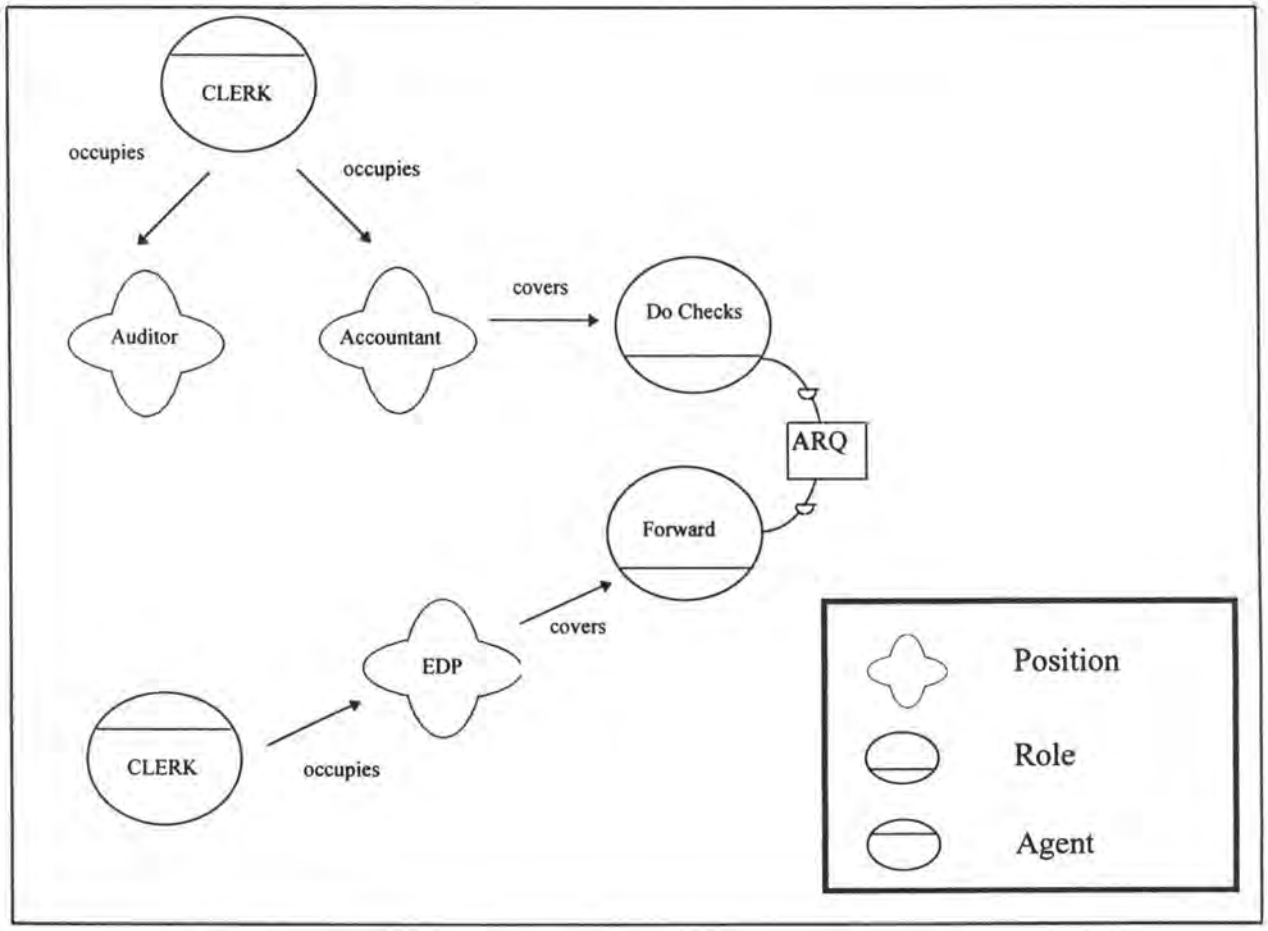


Figure 2.3. The Credit Card Company actor decomposition

Figure 2.3. describes the position and role occupied and played by the clerk agents in our Credit Card Company example.

A clerk agent occupies, for example, the position of a chartered accountant or an auditor. We assume that the different verifications are made by the chartered accountant. By analyzing Figure 2.3., we can find out that inside the Credit Card Companies' organization, the chartered accountant depends on the ARQ message it gets from the Host in order to execute its role i.e. in order to execute the '*Do Checks*' role.

By analyzing an agent, its position and role, the analyst can find out whether the dependency is linked to the agent or to the position it occupies. If we assume that the agent is able to execute the different verifications as it occupies a strategical position in the organization and that no particular knowledge is required in order to evaluate the incoming requests (the agent has only to apply a certain number of simple rules in order to find out whether the request can be accepted or not), the dependency is linked to the agent's position.

Degree of dependency

An existing dependency between two actors does not ensure the depender the achievement, the production or the execution of the dependum. The dependee may fail or may not be interested in bringing about the dependum.

In order to evaluate the implications of this non achievement, the i* framework makes the distinction between 3 different degrees of dependencies.

In an Open Dependency, the failure affects the depender but the depender can still achieve its goal. In a Committed Dependency, the failure can affect some parts of the depender and it can become difficult for the depender to achieve its goal. In a Critical Dependency, the depender' s goals become impossible to achieve in the case where the dependum is not brought about.

Figure 2.4. describes the notations used in order to represent the different types of dependencies. An 'X' symbol is used to characterize a Critical Dependency and a 'O' symbol to represent an Open Dependency. If no label is added to the dependency, the dependency is considered as a Committed Dependency.

Let us illustrate the different degree of dependencies by analyzing the *Pay Article* task dependency existing between the customer and the Credit Card Company.

If the price of the item is too high and the customer can not afford to buy it, the customer's behavior can be affected but the customer can still achieve its goal (i.e. get in possession of the item) for example by leasing it. The existing dependency is in this case represented by an Open Dependency.

A Committed Dependency can be illustrated by taking the assumption that the leasing of the item or other mechanisms that allow the customer to get into possession of the item are not possible. As the customer can not get in possession of the desired item, its behavior may be highly influenced.

In the case where the item represents an essential foodstuff for the customer i.e. a good necessary to ensure its living (like water, bread, meat ...), the fact that the item can not be bought can have a considerable effect on the customer's survival. The dependency is represented in this case by Critical Dependency.

In order to find out if dependencies may fail in a given organization or process, the concepts of Enforcement, Insurance and Assurance have been introduced in the i* framework.

Enforcement, Insurance and Assurance

A dependency is enforced, if a reciprocal dependency exists between two actors. In our example, such a reciprocal dependency exists between the shopkeeper and the Credit Card Company. The company depends on the shopkeeper to get accurate transaction information.

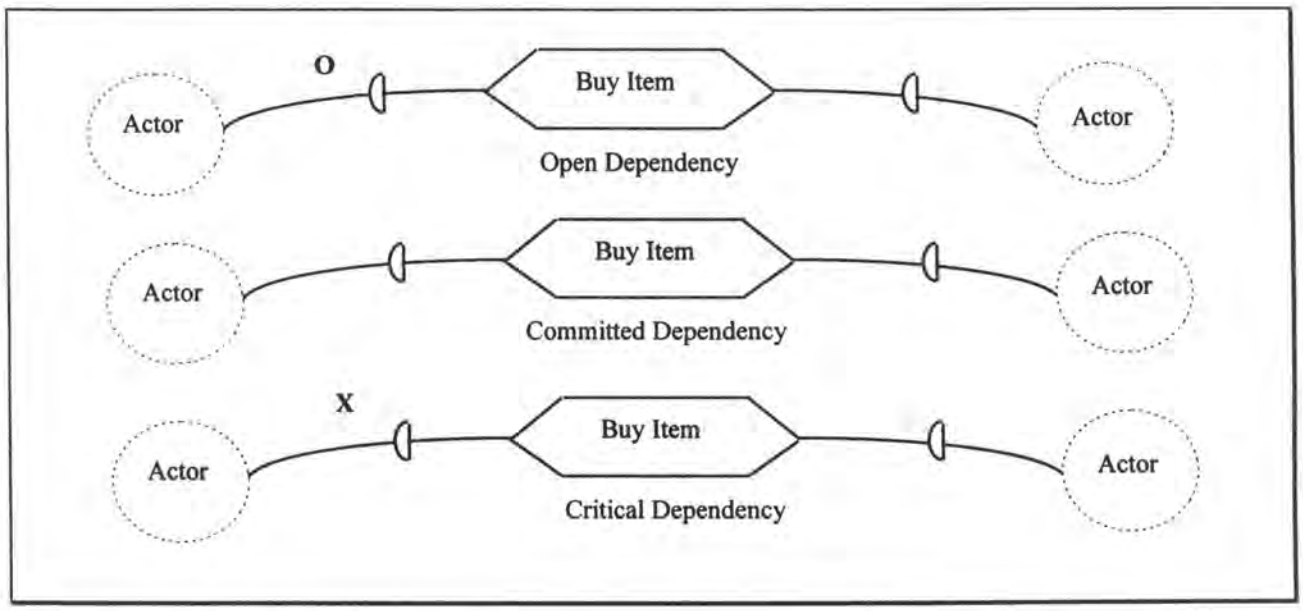


Figure 2.4. Notations used for the different degrees of dependency

As we have mentioned previously, the Credit Card Company's vulnerability is soften by the fact that the shopkeeper depends on the company to get paid.

The notion of Insurance is used in order to describe a situation in which an actor has the choice of several dependees so that its chances to get the dependum achieved, produced or executed are better. A customer can, for instance, have several shops at its disposal where it can purchase a particular item. If a shopkeeper does, for instance, not accept the customer's credit card, the customer can still leave the shop and purchase the item in a shop where its credit card is recognized as a means of payment.

The last situation in which a depender can be quite optimistic to get its dependum brought about is described by the Assurance concept. The chances that a dependum is brought about are good, as the dependee itself is also interested in the achievement, the production or the execution of the dependum.

In our example, the shopkeeper can be quite optimistic in order to get the Credit Card Companies' response as the company tries to maximize its profit by satisfy as much as possible its customers.

The Strategic Rationale Model

Definition

The aim of the Strategic Rationale model is to describe the components of a process or organization, their links as well as the rationales behind them.

The basic structure of the Strategic Rationale (S.R.) model is provided by the Strategic Dependency (S.D.) model. By specifying the internal behavior of each actor, the Strategic Rationale model describes how a certain dependum represented in the Strategic Dependency

model is achieved, produced or executed. Two new links are introduced by the Strategic Rationale model in order to describe this achievement, production or execution : the Task decomposition link and the Means-ends link.

As several internal behaviors are able to create the same dependum, the Strategic Rationale does not limit its description to one possible way of acting. Alternative ways allowing to bring about the same dependum can be represented by using the Strategic Rationale model. As the different alternatives can have different implications for a given actor, the positive and negative contributions of each alternative can also be expressed.

The next two paragraphs (sub-sections) describe the task decomposition link respectively the means-ends link. The different i* notions are again illustrated by examples taken from our Credit Card Purchase example. The Strategic Rationale model corresponding to the Credit Card Company example is depicted by Figure 2.5.

1. The Task Decomposition Link

In a task dependency, the depender depends on the dependee to get a certain condition or state achieved. The difference with a goal dependency consists in the fact that a task is accompanied by a description of how the condition or state has to be achieved.

A task decomposition link allows us to describe how a task has to be realized by specifying the different subcomponents that have to be executed, produced or achieved. Four different sub-components can be used corresponding to the four basic i* components: the task, the goal, the resource and the softgoal component. Depending on the type of the subcomponent, the sub-component is called a subtask, a subgoal, a ResourceFor or a SoftgoalFor component. The different graphical representations are given by Figure 2.6. and are analyzed here afterwards.

a. Task into Task decomposition

A Task into Task Decomposition allows us to describe how the main task has to be executed by specifying one or several subtasks. Each of those subtasks allows to achieve a certain subcondition or substate. The main or parent task is achieved by executing the subtask(s) and their respective subcomponent(s).

The i* framework describes the subtasks that have to be executed, but does not describe the order in which they have to be executed. This is due to the fact that the i* framework recognizes each actor a certain freedom of acting. This liberty has already been underlined in the previous section describing the goal dependency. In a Task into Task decomposition, the order in which the different subtasks are executed is established by the actor. In the real world i.e. at run-time, the actor is however often forced to respect a certain order so that a certain dependum can be brought about. If the reader wants to find out the order in which a particular actor has to be execute the different subtasks, the reader must possess a certain knowledge of the described process or organization. This can however raise some difficulties especially when the reader is faced to a complex process or organization.

In a Task into Task decomposition, a main or parent task is decomposed into subtasks. These subtasks can then theoretically be decomposed again and again. Each new level provides new

information about the components that have to be achieved in order to achieve the component located at the parent level.

This decomposition approach is used by many specification languages among others also by the Albert II language described in the previous chapter. In the Albert II language, for instance, an action can be decomposed into subactions which, at their turn, can be decomposed again. The decomposition usually ends when the analyst has reached a level containing atomic actions i.e. actions which can not be further decomposed.

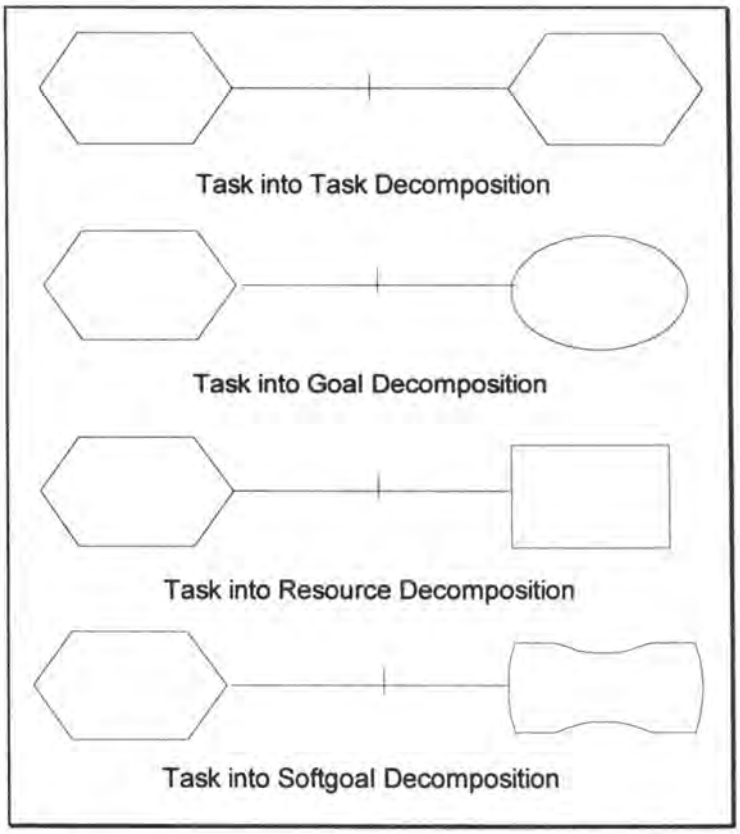


Figure 2.5. Graphical representation of a task decomposition link

In the i* framework, the decomposition process usually ends when no more further subcomponents can be found that represent a strategic importance for an actor. There may situations exist in which a task could be further decomposed but the analyst has stopped the decomposition process as the new components do not represent a strategical interest for the described actor.

In our Credit Card Purchase example, the Credit Card Company has to perform a certain number of checks before the customer's transaction request can be accepted. First, it has to verify the validity of the received credit card number. The customer's balance has then to be checked i.e. the company has to find out if the customer has not exceeded its maximum debt level for the actual period. Finally, the company has to find out if the customer's expenses have not exceeded a certain amount represented by the Transaction History limit.

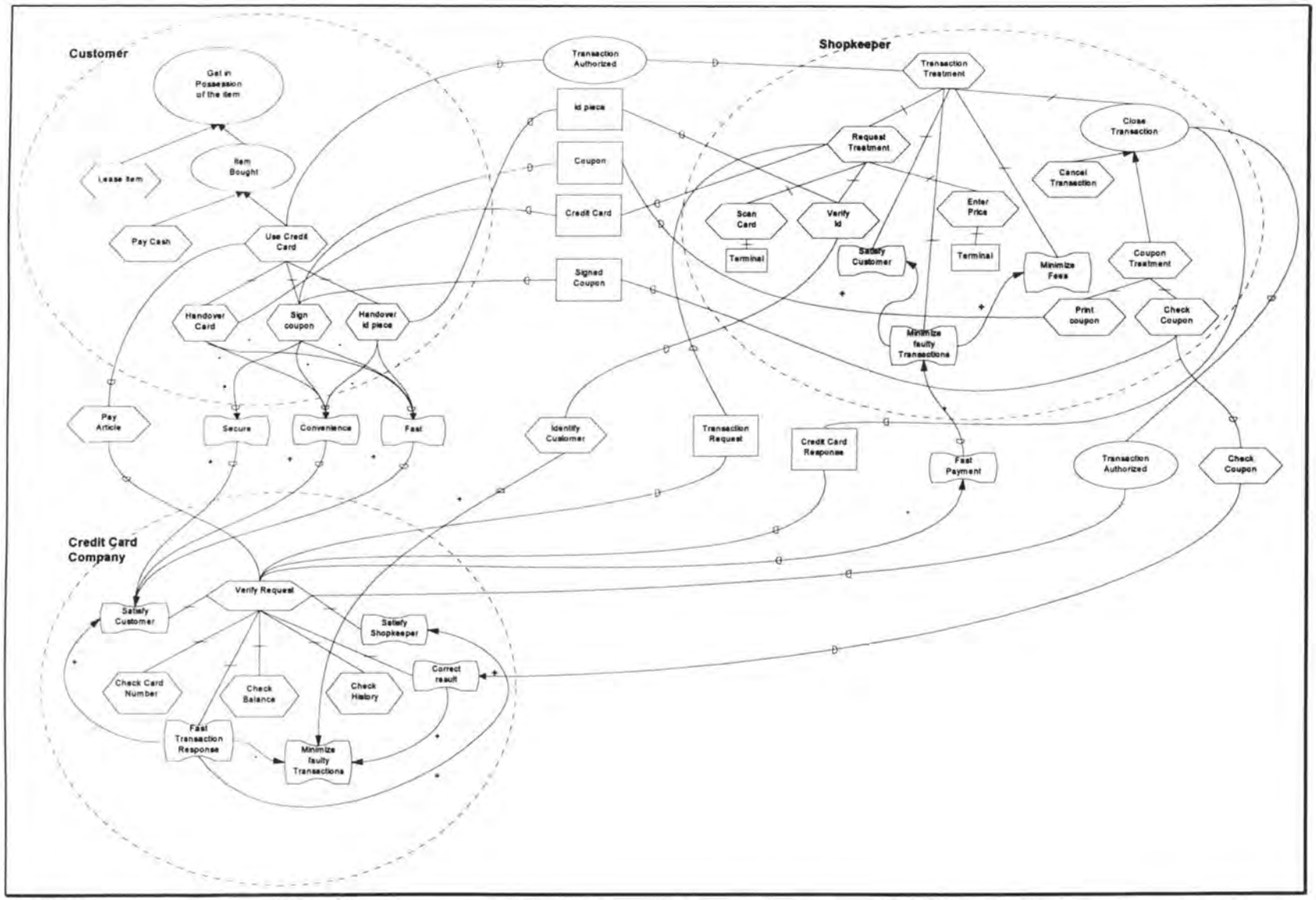


Figure 2.6. The Strategic Rationale model of the Credit Card Purchase example

Figure 2.7. describes that the Credit Card Company has to perform an *internal* task called *VerifyRequest* in order to realize the state or condition described by the *Transaction Authorized* goal. The *VerifyRequest* task can be decomposed into three subtasks. Each subtask represents one of the three checks that the company has to perform. An *internal* task means that the executed task belongs to the actor's internal behavior and that the fact that the task or subtask is executed is not directly perceived by the other actors (in opposition to the task's effects).

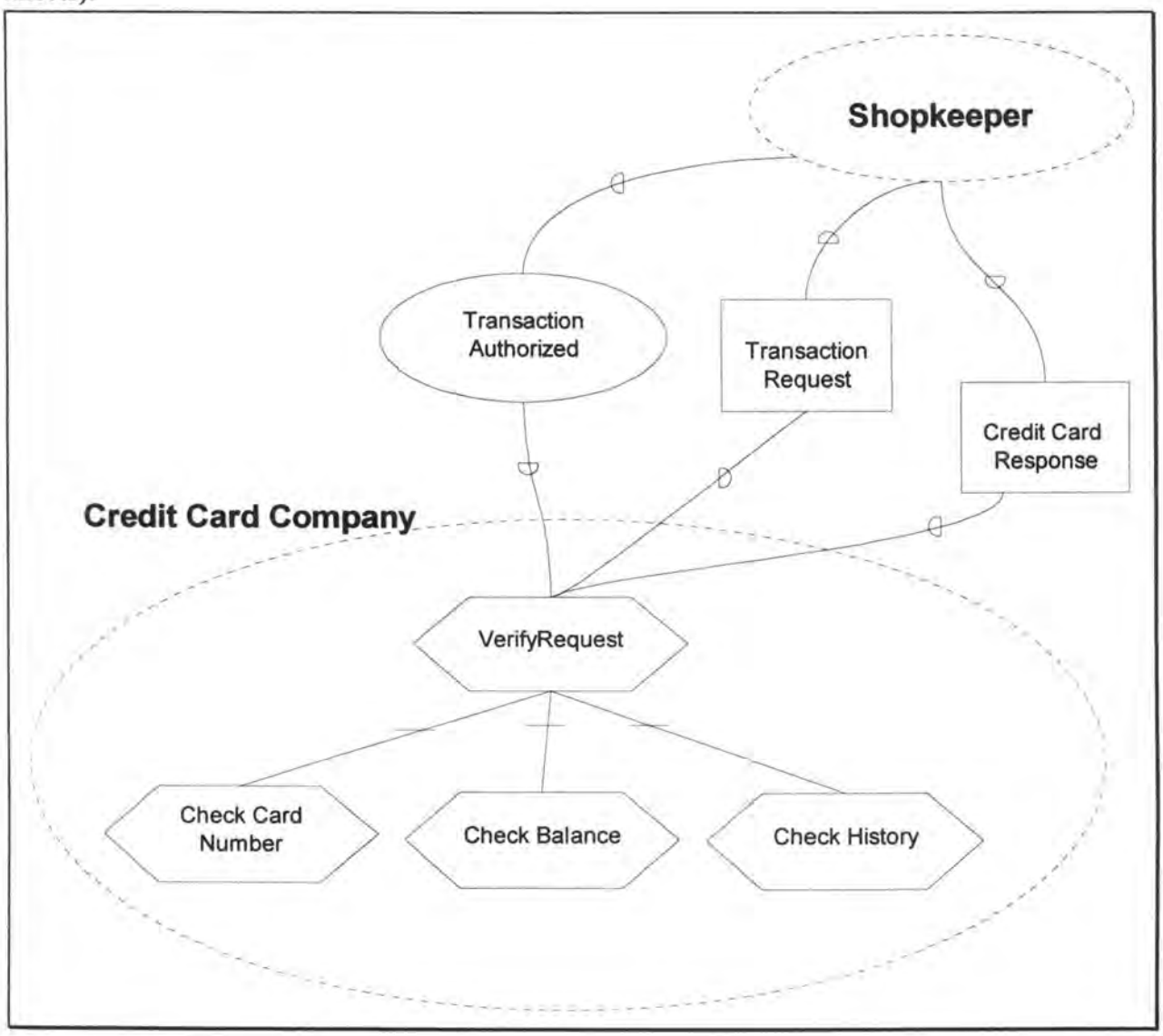


Figure 2.7. Example of a Task into task decomposition

b. Task into Goal Decomposition

In a Task into Goal decomposition, indications on how a task has to be performed are given by specifying a subcondition or substate that has to be achieved. How this substate or subcondition has to be achieved is not directly specified and alternatives may be taken into account.

Figure 2.8. describes that in order to find out if a customer's transaction request can be accepted or not, the Credit Card Company has to execute two checks (*CheckCardNumber* and *CheckHistory*) and achieve the '*Balance Checked*' goal. If we assume now that the Computer Department is in charge of executing the different checks, the Computer Department has the choice between two alternatives.

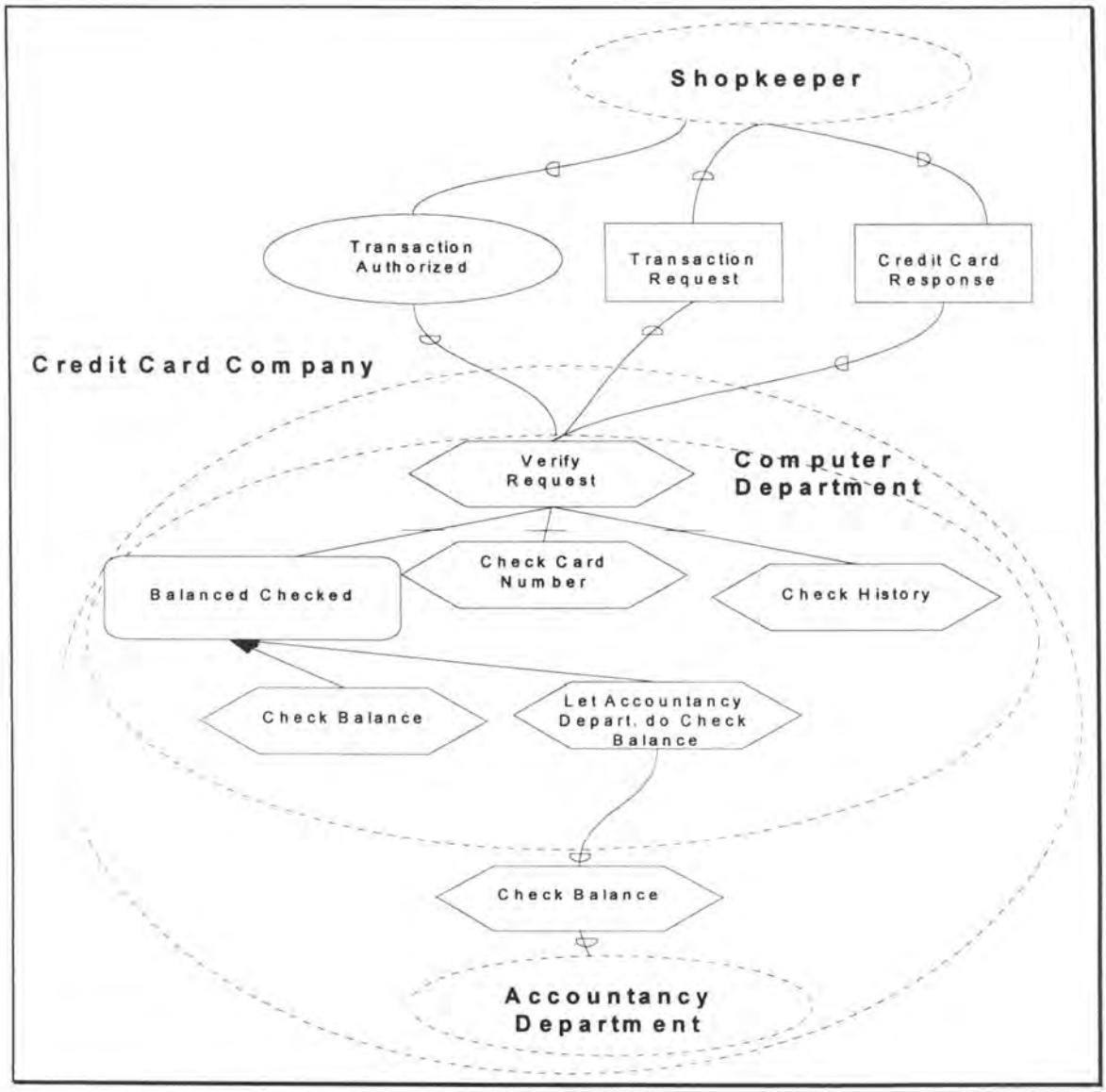


Figure 2.8. Task into Goal Decomposition

First, the Computer Department can chose to execute the '*Check Balance*' task by itself. Second, it also has the possibility to ask the Accountancy Department to perform the check. The two arrows represent two means-ends links and are used to describe the fact that the agent has several ways at its disposal which allow the achievement of a certain goal. For further details about means-ends links, please consult the next subsection analyzing the concept of means-ends links.

c. Task into Resource Decomposition

A Task into Resource Decomposition describes a situation in which a resource is needed in order to execute a task. Let us remark that a resource is considered by the i* framework as non-critical and that the only problem related to a resource is whether it is available or not. The resource is however only specified in the Strategic Rationale model if it represents a strategic interest for the described actor.

Figure 2.9. describes that in our Credit Card Purchase example, the shopkeeper depends on the customer's credit card (external resource) in order to execute the *RequestTreatment* task. The *RequestTreatment* task is decomposed into a sub-task called *ScanCard* which uses the card reader of the terminal (internal resource).

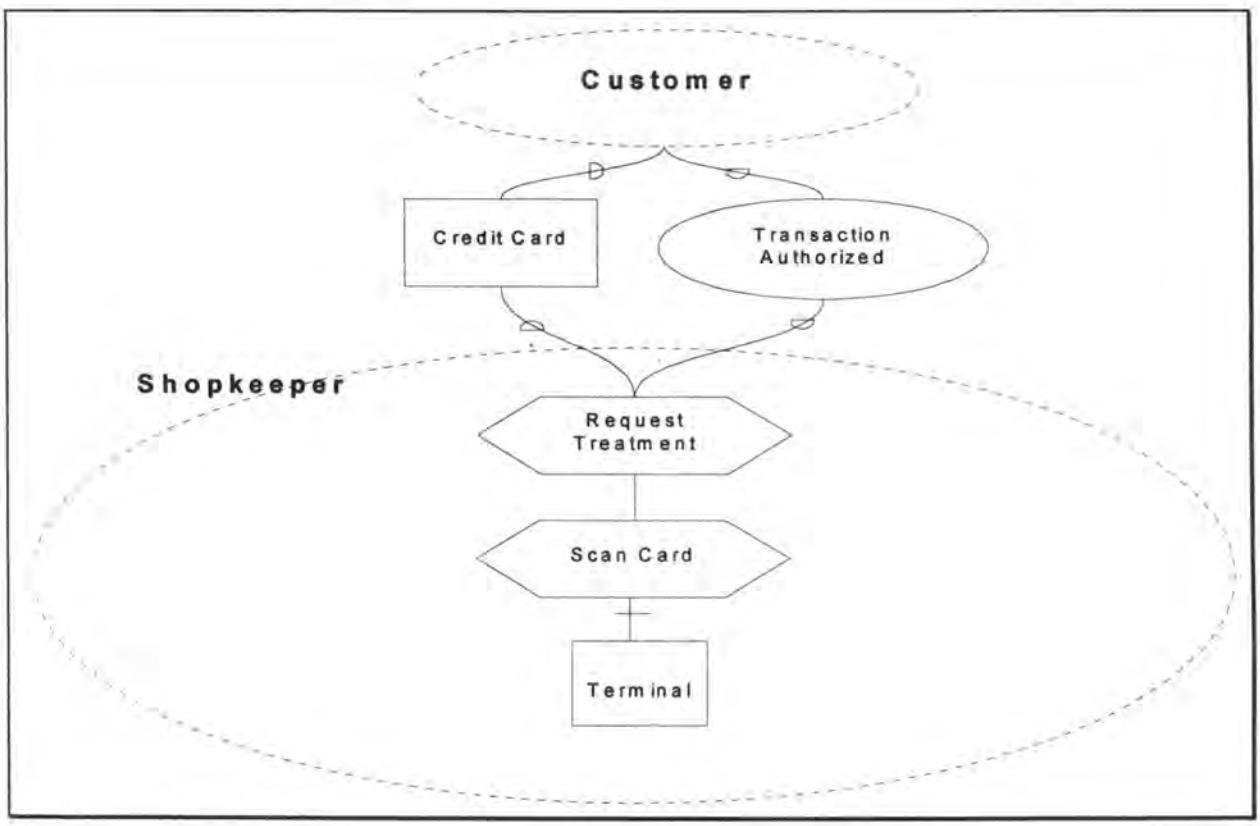


Figure 2.9. Task into Resource Decomposition

d. Task into Softgoal decomposition

By decomposing a task into a SoftgoalFor component, the i* framework allows us to specify a non-sharpened condition that a particular actor considers as important and which has to be satisfied by the decomposed task. The non-sharpened condition is again subject to interpretations. The condition described by the softgoal can however be used as a criteria which has to be satisfied while decomposing a component into sub-components.

Figure 2.10. represents a situation in which the *VerifyRequest* task is decomposed into a subtask, two subgoal and one softgoal subcomponent. The softgoal subcomponent specifies that the actor i.e. the Credit Card Company considers a fast response as important.

In the case where several alternative ways exist which allow an actor to achieve the same state or condition, the softgoal criteria can be used in order to chose one of the existing alternative ways. How alternatives are represented is described in the next section.

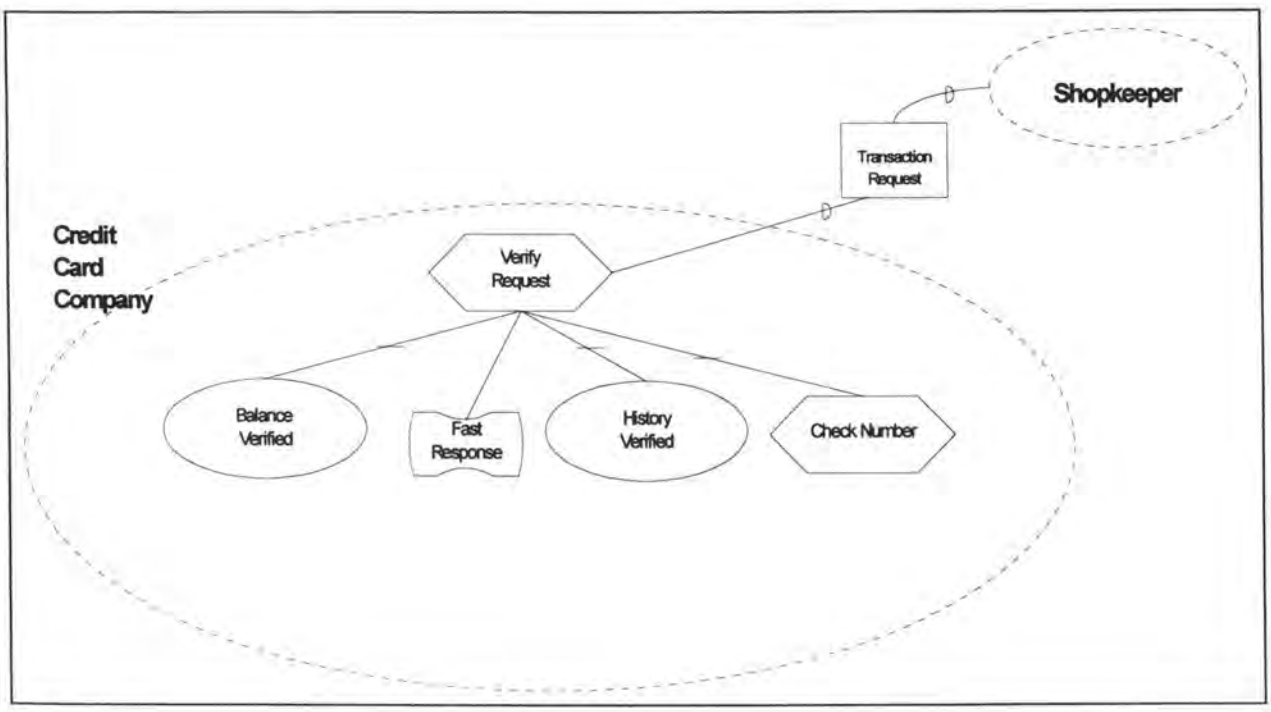


Figure 2.10. Task into Softgoal Decomposition

2. Means-ends links

Definition :

> "A Means-ends Link indicates a relationship between an end - which can be a goal to be achieved, a task to be accomplished, a resource to be produced, or a softgoal to be satisfied - and a means for attaining it. The means is usually expressed in the form of a task, since the notion of task (...) embodies how to do something."

Graphically, a means-ends link is represented by an arrow where the arrow's source describes the means and the arrow's end the end.

Figure 2.11. is derived from Figure 2.10. and illustrates a Goal-Task link (GT Link). Two different alternatives exist to achieve the '*Balance Verified*' goal . The Computer Department may delegate its responsibilities to the Accountancy Department or perform the task by itself.

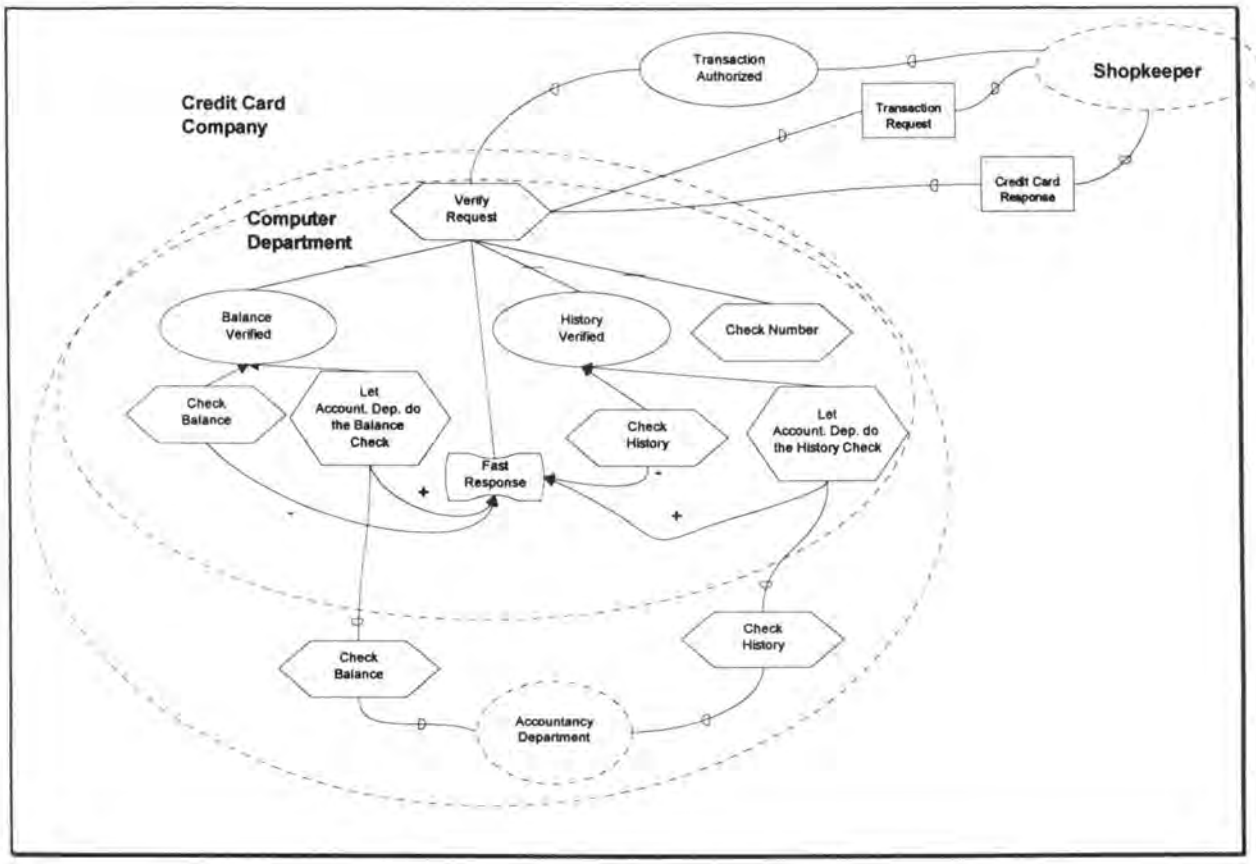


Figure 2.11. Means-Ends links

In a Resource-Task link (RT Link), a resource is produced by the execution of a certain task. Figure 2.11. describes the fact that a message containing the Credit Card Companies' response is created by the execution of the '*VerifyRequest*' task. Let us however stress that in this particular example, the RT Link is not represented by an arrow but by a dependency link. This, because the created resource '*Credit Card Response*' represents a dependum.

Two special means-ends links are introduced by the i* framework implicating softgoals.

A Softgoal-Task Link (ST Link) expresses the fact that a certain task contributes to the realization of a certain softgoal (represented by the end). A label is added to the arrow informing the reader that the means task contributes positive ('+') or negatively ('-') to the realization of a certain softgoal.

In Figure 2.11., the '*Let Accountancy Department do the Balance Check*' task contributes positively to the realization of the '*Fast Response*' softgoal (we assume that the Accountancy Department can perform the task faster as it has all the necessary information at its disposal) whereas the execution of the '*Check Balance*' task by the Computer Department, contributes negatively to it (we assume that an operator from the Computer Department has to move from the Computer Department to the Accountancy Department in order to execute the verification task).

Softgoals also can contribute positively or negatively to the realization of other softgoals. If we add a second softgoal, called '*Correct Result*' softgoal, to the task decomposition described

by Figure 2.11., the new obtained verification process, depicted by Figure 2.12., must be fast and a correct result must be obtained. We say that the '*Fast Response*' softgoal contributes negatively to the '*Correct Result*' softgoal as the risk of an incorrect result increases in the case where an actor tries to reduce the time of the verification process.

Two additional means-ends links can be used in a Strategic Rationale model i.e. the Task-Task Link (TT Link) and the Goal-Goal Link (GG Link). Both links are however seldom used in a Strategic Rationale model.

A TT Link expresses the fact that the execution of different alternative tasks may allow the realization of the task represented by the end. A GG Link describes a case in which a condition or a state can be achieved by achieving a subcondition or substate. Again, different alternatives may exist which allow us to achieve the state or condition represented by the means or by the end.

3. Analysis of the Strategic Rationale Model

Routine

Definition

> "A routine is a sub-graph in the Strategic Rationale graph with a single link to a "means " node from each " end " node. "

A routine describes a single way to achieve a certain condition or state described by an end. A routine implies that a certain number of choices are made by the analyst at each or-node as a routine may not contain alternatives.

A possible routine for the Credit Card Companies' verification process is described by Figure 2.13. It describes that in order to get the customer's transaction request validated, the two checks are shared between the Computer Department and the Accountancy Department.

As a routine is a sub-graph of the Strategic Rationale graph and as the Strategic Rationale graph is partly based on the Strategic Dependency, external dependencies may be contained in a routine.
In order to underline the positive and / or negative contributions of a certain routine, multiple means-ends links containing softgoals may be allowed.

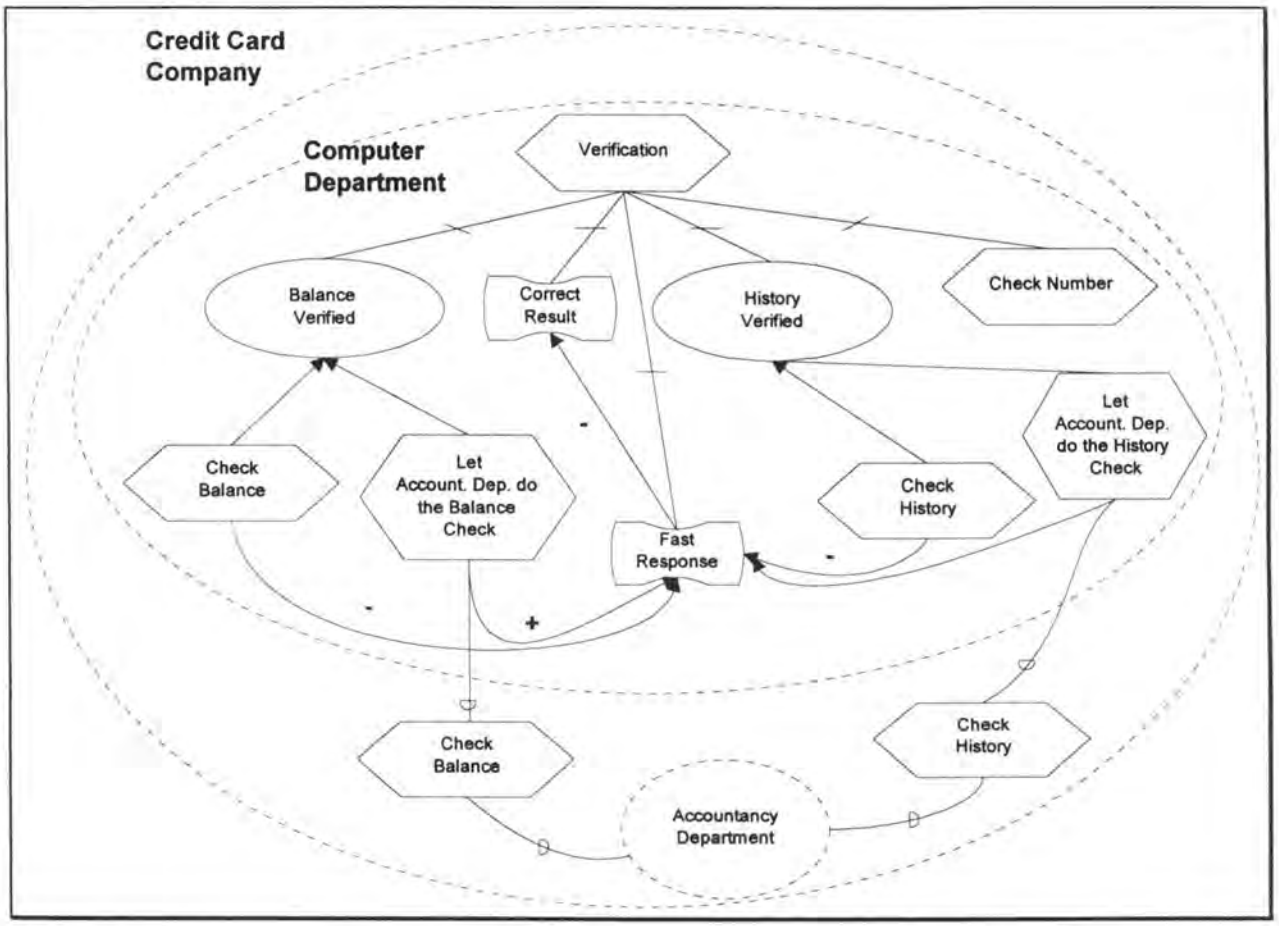


Figure 2.12. Adding an additional softgoal to the Verification task decomposition

Rules

A routine represents a certain way of achieving a certain state or condition. It can be predefined inside an organization or process i.e. can be obligatory for all the different actors.

Definition

"A rule consists of an applicability condition, a means and an end. "

Based on this definition, we can say that "a means-end link is an application of a rule in a context in which an actor believes that the applicability condition will hold".

A rule, for instance, consists in saying that an employee from the Computer Department of the Credit Card Company can execute the '*Check Balance*' task of the Credit Card Companies' verification process, if it has the necessary knowledge needed to execute the task.

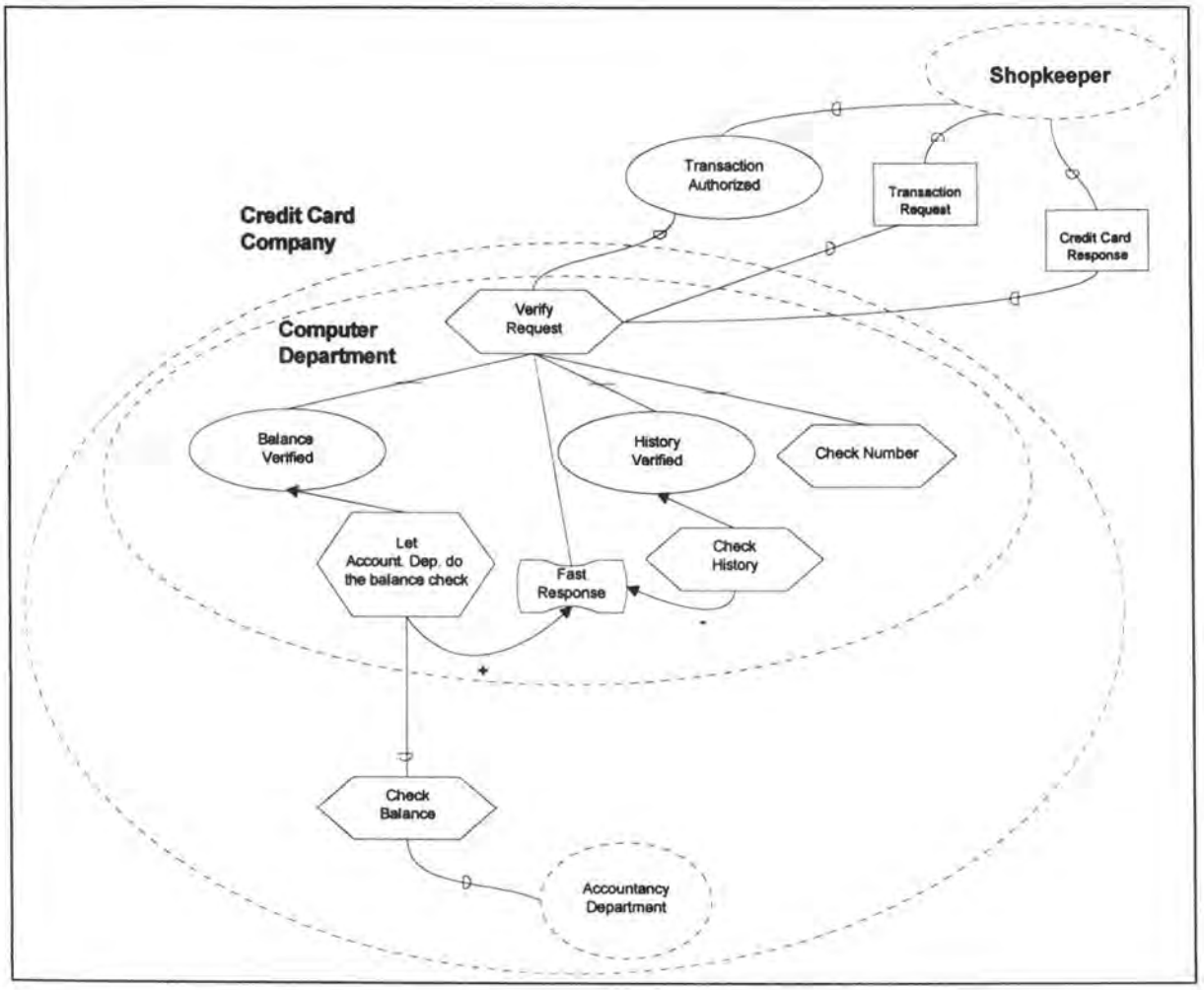


Figure 2.13. Routine derived from the 'Verification process'

Belief

For a given rule, an actor has to evaluate if the applicability condition is or will be satisfied and act in consequence. A clerk from the Credit Card Company can estimate or believe that it is not able .i.e. does not possess the required knowledge or qualification to execute a particular task.

As we mentioned earlier, the existence of a dependency does not ensure the dependee that the dependum is produced, achieved or executed. The dependee does not have the time or is not interested in bringing about the dependum.

Another possibility why a certain dependum is not brought about can consist in the fact that there exist no routine which allows a certain actor to achieve a certain state or condition or that the actor believes that the applicability condition will not hold.

The choice, whether the dependee tries to achieve a certain state or condition or not, is a personal choice and can be influenced by the positive and / or negative contributions of a given routine, the knowledge of the actor as well as its experience.

The concepts of a routine, a rule and an actor's belief give us important information about a certain actor and allow us to foresee, to a certain degree, its behavior. We can however never be completely sure whether an actor is or will be able to achieve a certain condition or state and which routine is or will be chosen to achieve the requested condition or state.

Ability, Workability, Viability and Believability

The Strategic Rationale model allows us to represent the internal behavior of a certain actor inside an organization or process. In opposition to conventional process models where a process is represented by a set of actions (which are connected by flows), the Strategic Rationale model allows us to go further in the analyze of a given process.

Whereas conventional process models allow us to represent the "whats" of a given process, the Strategic Rationale model allows us to represent the answers to three different kinds of questions :

how : how does an actor achieve a certain condition or state ?

for instance : how can the '*Verify Request*' condition be achieved at the Credit Card Companies' level ?

why : why does a certain actor have a certain interest in achieving a certain condition or state ?

for instance : why does the Credit Card Company delegate its responsibilities to the Accountancy Department ?

how else : how else can an actor achieve a certain condition or state ?

for instance : Does another routine exists which allows the Credit Card Company to get the transaction request be verified ?

The answers to the " how" questions are obtained by analyzing the different means-ends links and task decompositions of the Strategic Rationale model or by analyzing the different existing routines. The positive and negative contributions of a certain process allows us to find out why a particular actor is interested in the achievement of a particular state or condition. Finally, the "how else" questions can be answered by analyzing the different means-ends representing the different existing alternatives.

While these questions allow us to get a better understanding of a given actor, a second set of questions, especially of interest for a strategic analysis, allow us to find out process level based information :

Ability : Does a process have a particular routine which allows an actor to achieve a certain state or condition ?

Workability : Does an actor possess a certain routine that allows it to achieve a certain state or condition ?

Viability : To which extent are the different actor's softgoals satisfied by a given process ?

Believability : What evidence are there to confirm or disconfirm that a certain process will work ?

By asking and analyzing those two sets of questions, a reader can find out certain information about a process and the behavior of a particular actor that can not be obtained or can only be obtained with certain difficulties by using conventional representation models. A simple fact that underlines the advantage of the Strategic Rationale model and in more general the use of the i* framework.

Chapter 3 : Analysis of the existing links between the i* framework and the Albert II language

Section 1. Introduction

From its beginning on, the computer domain and all its related components have seen themselves expanding at an incredible speed. Soon, analysts found out that the only way allowing them to realize important and complex systems consisted in following a certain structured way of proceeding.

Whereas the first methodologies only described in a simplified way how systems have to be realized, the following ones becomes more and more complex. In order to show their evolution, let us analyze one of the first theories : the Waterfall model introduced by Boehm [6].

The Waterfall model describes the basic steps that an analyst has to follow in order to realize a certain application or system. Based on the Waterfall model, three different stages or steps have to be executed : (i) the specification stage, (ii) the design stage and (iii) the implementation stage.

During the first step, the customer's desires and wants are written down in a document called requirements document. The design module takes these requirements in the second stage as basis for its work and specifies the design of the system. The obtained design is then used in the next stage in order to implement the system. The Waterfall model specifies the order in which the different modules have to be executed. The analyst may however go back to a previous stage in the case where problems are detected at a certain level.

If we focus now on the specification stage, a certain evolution can also be detected. Whereas the first specification languages have been particularly designed for a restricted and simple domain, the following ones have evolved and can be applied today to a various number of different and complex domains.

An example of these early specification languages is for instance the SADT language introduced by Ross & Schoman [7].

The SADT language is composed of boxes representing actions and different types of arrows. An arrow is linked to a box and represents an input, an output, a control or a resource used by the action.

The advantage of an SADT model consists in the fact that it can be easily used by non specialists in order to represent systems composed of activities and flows as the only used components are boxes and arrows. The SADT model also allows the decomposition of a

complex system into less complex sub-systems. The obtained sub-systems can then be decomposed again and again.

One of its disadvantages however consists in the fact that when a complex system is decomposed, the number of obtained levels may become very important. An important number of levels however makes the understanding of the model more difficult.

A second desadvantage of a SADT model consists in the fact that it describes a particular system at a given moment by specifying its activities and flows. Constraints related to those activities and flows can not be represented by the SADT framework. The concepts similar to the notion of an agent or an actor are also not included in the SADT framework.

Certain of those features are implemented in more recent developed languages like for instance the Albert II language described in Chapter 1.

Another evolution that took place and which affect the specification process is related to the fact that more and more existing applications and systems have to be improved. Most of the specification languages containing the above mentioned features are limited when being used in the reengineering process.

Their limitation consists in the fact that the obtained models only describe **WHAT** the different agents or actors are or have to do and not **WHY** they are or have to do it. The WHY however gives important information about a given application or system. These information are crucial for the reengineering process. And it is at this point that the i* framework steps in.

As we have seen in the second chapter, the i* framework is used to describe the dependencies that may exist between the different agents as well as the internal behavior of the different actors necessary for executing or achieving the dependum i.e. the source of dependency.

Whereas the Albert language allows us to describe *what* has to be achieved by specifying the vocabulary and constraints of a given system, the i* framework allows us to represent *why* a particular way of achieving a dependency represents a certain interest for a particular agent.

The combination of the Albert and the i* framework allows the analyst to draw models which specify both the why and the what of a given application or system and makes the analyst's job more easier, especially in the reengineering process.

Figure 3.1. describes, by using a time evolution axis, how both frameworks can be used together in order to get a valid specification of a system. As you may observe, the i* framework is used rather at the beginning of the specification process whereas the Albert language only starts at the middle of the time evolution axis.

The reason why both frameworks are used at different moments can be explained by the fact that when an analyst starts writing an Albert specification, the analyst must already have made some preliminary work.

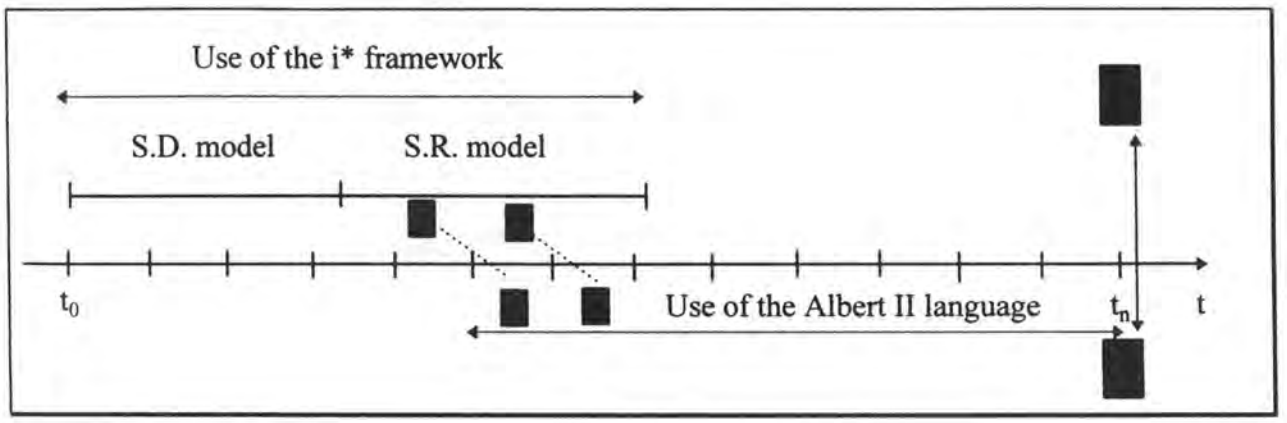


Figure 3.1. Use of both frameworks

This preeliminary work consists in analyzing and understanding the system that has to be realized as well as the domain in which the futur system will be implemented. Stated otherwise, the analyst must already have a certain number of ideas in mind about how to achieve the different requested wants and needs before creating the system's specification.

Once the different ways of acting have been found out, the analyst's job consists in evaluating these different ways and in chosing one of them. This way of acting is then represented at the Albert level.

As the i* framework allows us to represent and evaluate different alternatives allowing to achieve the same sub-condition or sub-state, the Albert work can be based on the models previously created at the i* level so that the analyst's work becomes a lot more easier.

The Strategic Dependency model for instance allows us to describe the existing dependencies between the different agents and arrange their relations in a way that the dependencies are well distributed. Once the dependencies arranged and the Strategic Dependency model drawn, the Startegic Rational model can be created. It describes the internal behavior of the different agents from a intentional and strategic viewpoint as well as the possible existing alternatives that allow to achieve the different dependums.

On the basis of the described alternatives and their evaluations, a final way of acting, called routine at the i* level, is chosen. It describes the way how the main condition or state has to be achieved by enumerating a set of tasks, goals, resources and softgoals that have to be executed, achieved or produced. A particularity of a routine consists in the fact that it describes one and only one way to achieve a certain state or condition. Stated otherwise, a routine does not contain alternatives.

Once a final routine has been chosen, the analyst may start creating the corresponding Albert specification. The aim of the Albert language is to describe the possible actions and the related constraints that may occur during the life-cycle of the different agents and which allow us to achieve the conditions or states described in the corresponding i* models.

At the beginning of the Albert specification process, the analyst starts by analyzing the different components of the Strategic Rational model and tries then to represent them by an

Albert fragment. Progressing in time, the different Albert fragments are arranged together and form finally a unique Albert specification. Whereas at the beginning of the Albert process only links between the i* and Albert fragments existed (the links are represented by a doted line in Figure 3.1), the final obtained models of both frameworks (at time period t_n) describe the same application or system but at different levels.

The analyst's work can be considerable simplified if both framework are used together during the specification process. In the case where only the Albert language is used, the analyst has to find out the different tasks that have to be executed by the different agents as well as the different existing alternatives, chose a certain way of acting, decompose that way into sets of actions and finally specify each of them in order to get a valid Albert specification.

The i* framework allows us to simplify the analyst's work at the Albert level. The Albert language at its turn may be used in order to check whether a particular i* model can be implemented or not.

In the case where an i* model is specified at the Albert level and problems are detected, the analyst can go back to the i* level and try to solve the problem by searching, evaluating and chosing a new alternative. The process of going back and forth between the i* framework and the Albert language may be repeated several times until a solution has finally been found.

In the next sections, we analyze the existing links between the i* and the Albert language by adopting an *i* to Albert* approach. This means that we take the different components of the i* framework as basis (as the i* models are created before the corresponding Albert specification) and describe whether links to the Albert language exist. In the case where links between the 2 frameworks are detected, they are illustrated by different fragments (examples) taken from our Four Messages Protocol example introduced in Chapter 1.

Lets however stress the fact that an existing link between the Albert and the i* framework does not mean that an i* model can be easily *translated* by a certain prefixed way of acting. There exist no predefined algorithm or other mechanism which allows us, by applying a certain number of rules, to receive a valid Albert specification based on a i* model. The analyst has still to possess a certain knowledge about the related domain in order to find out and specify the different Albert components and particular the different existing constraints.

The structure of this chapter is as following. Section 1 introduces the i* models of the *Four Messages Protocol* (4MP) example introduced in Chapter 1. Section 2 analyzes the task dependency and its decomposition. Section 3 describes the existing links between a goal dependency (and its decomposition) and the components of the Albert language. Section 4 and 5 analyzes the resource respectively the softgoal components (and their decompositions). Finally, Section 6 closes this chapter by taking a short look at the notion of *liberty of acting*.

Section 1. The i models of the Four Messages Protocol example*

The aim of this section consists in describing the obtained i* models representing the *Four Messages Protocol* example. For a complete description of this example, please refer to Chapter 1. As the Strategic Rationale model does not fit on one page, the model is decomposed into three submodels ; each submodel representing one of the three actors.

The Strategic Dependency model

The Strategic Dependency model of our 4MP example contains 3 actors : the terminal, the Host and the Bank actor. The Strategic Dependency model of our Four Messages Protocol example is given by Figure 3.2. Here below, we analyze the different actors' dependencies :

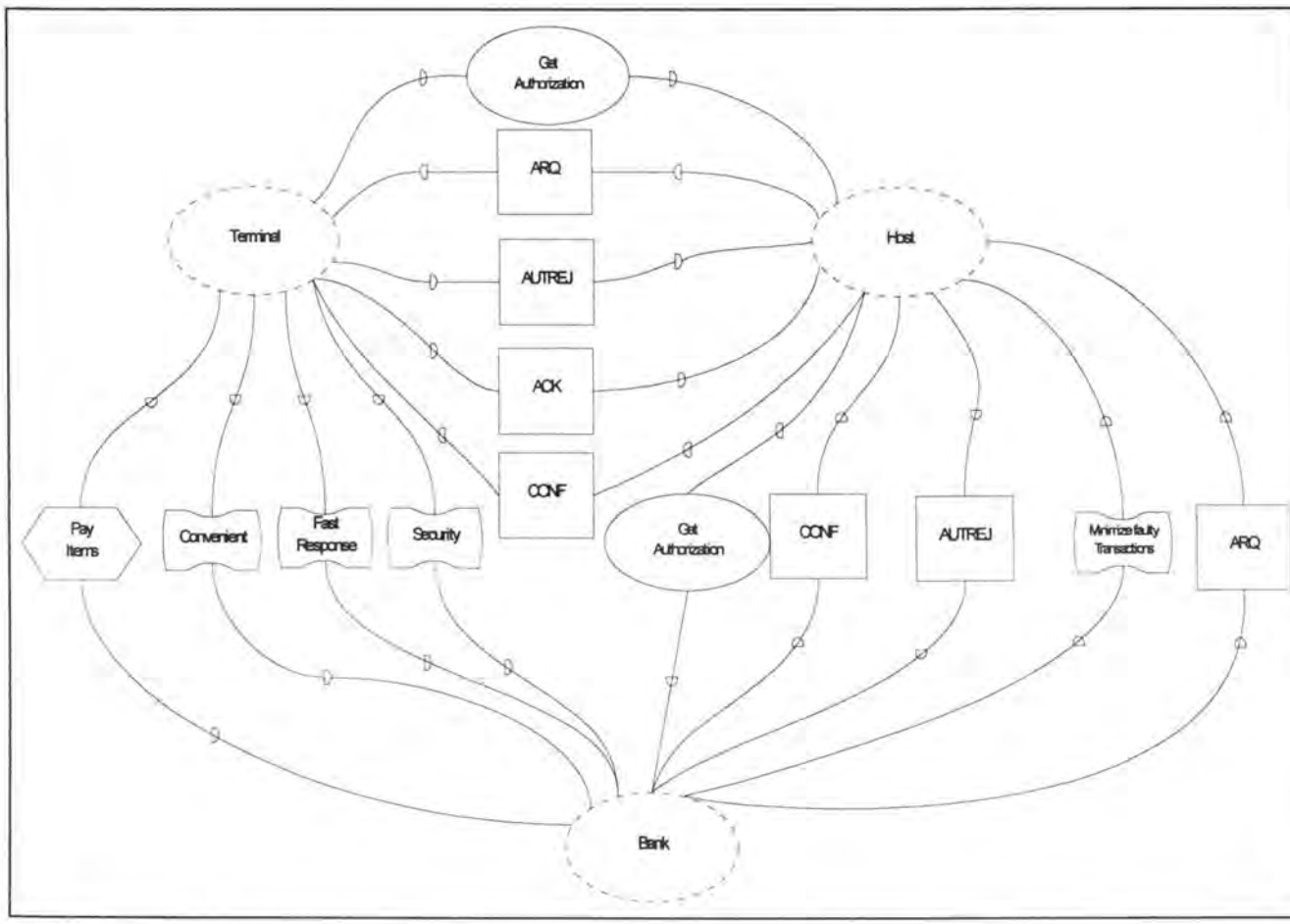


Figure 3.2. The Strategic Dependency model of the *Four Messages Protocol* example

a. The Terminal actor

The Terminal depends on the Bank actor in order to pay the bought goods or services. In our example, we assume that it is the bank which prescribes the different procedures that have to be followed by the different actors. Based on this assumption, the dependency is represented by a task dependency called *Pay Items*. We further assume that the Terminal actor has some needs and desires which are represented by three softgoals.

First of all, we assume that the Terminal does not want to wait too long before the transaction process is completed. This fact is represented by the *'Fast Response'* softgoal dependency. Second, we assume that the terminal is concerned about the *Security* of the made transaction. By a secure transaction we understand the security of the transaction information during the different exchanges as well as the fact that information are managed by the Host and the bank that allow to trace back the by-the-terminal made transactions in the case where problems are encountered. Finally, we take the assumption that the terminal requests a *Convenient* way of executing the transaction process. In our example, a convenient transaction represents a transaction where only a few interventions are requested from the terminal before the transaction is completed.

The terminal depends on the Host in order to get the bank's response whether the requested transaction has been accepted or not. We assume that the terminal is not interested in the way the Host has to act in order to get and transfer the bank's response so that we represent the dependency by a goal dependency called '*Get Authorization*' .

Finally, Figure 3.2 depicts the fact that the terminal depends on two resources provided by the Host. The respective dependums are the Authorization / Reject (AUTREJ) message and the Acknowledge Receipt (ACK) message.

b. The Host actor

The Host, at its turn, depends on two resources provided by the terminal. The-by-the terminal produced resources are (i) the Authorization Request (ARQ) message and the Confirmation (CONF) message. Figure 3.2. also describes the fact that the Host depends on the bank in order to get the bank's response. The dependum of the dependency is represented by the Authorization / Reject (AUTREJ) message. Again, we assume that the actor is not interested in the way the decision is taken and transferred so that the dependency is represented by a goal dependency called '*Get Authorization'*.

c. The Bank actor

The bank depends on the Authorization Request (ARQ) message as well as on the Confirmation (CONF) message it receives from the Host. The Bank also depends on the Host in order to minimize the fees related to the transaction process. In our example, we assume that the bank's fees are minimized by the accuracy of the received information. The more accurate the received information are, lesser complains occur and less transactions are cancelled by the customers. In our example, we assume that the number of cancelled transactions is in direct relation with the bank's profit. As the objective of minimizing the number of faulty transactions may be subject to interpretations (how many fault transactions are tolerated ?), the dependency is represented by a softgoal dependency called *Minimize Faulty Transactions*.

The Strategic Rationale model

a. The Terminal actor

In order to pay the bought goods or services, the terminal has to execute the *Transaction Treatment* task. The *TransactionTreatment* task is decomposed into a subtask called *RequestTreatment* and a subgoal called *ConfirmationRequested.*

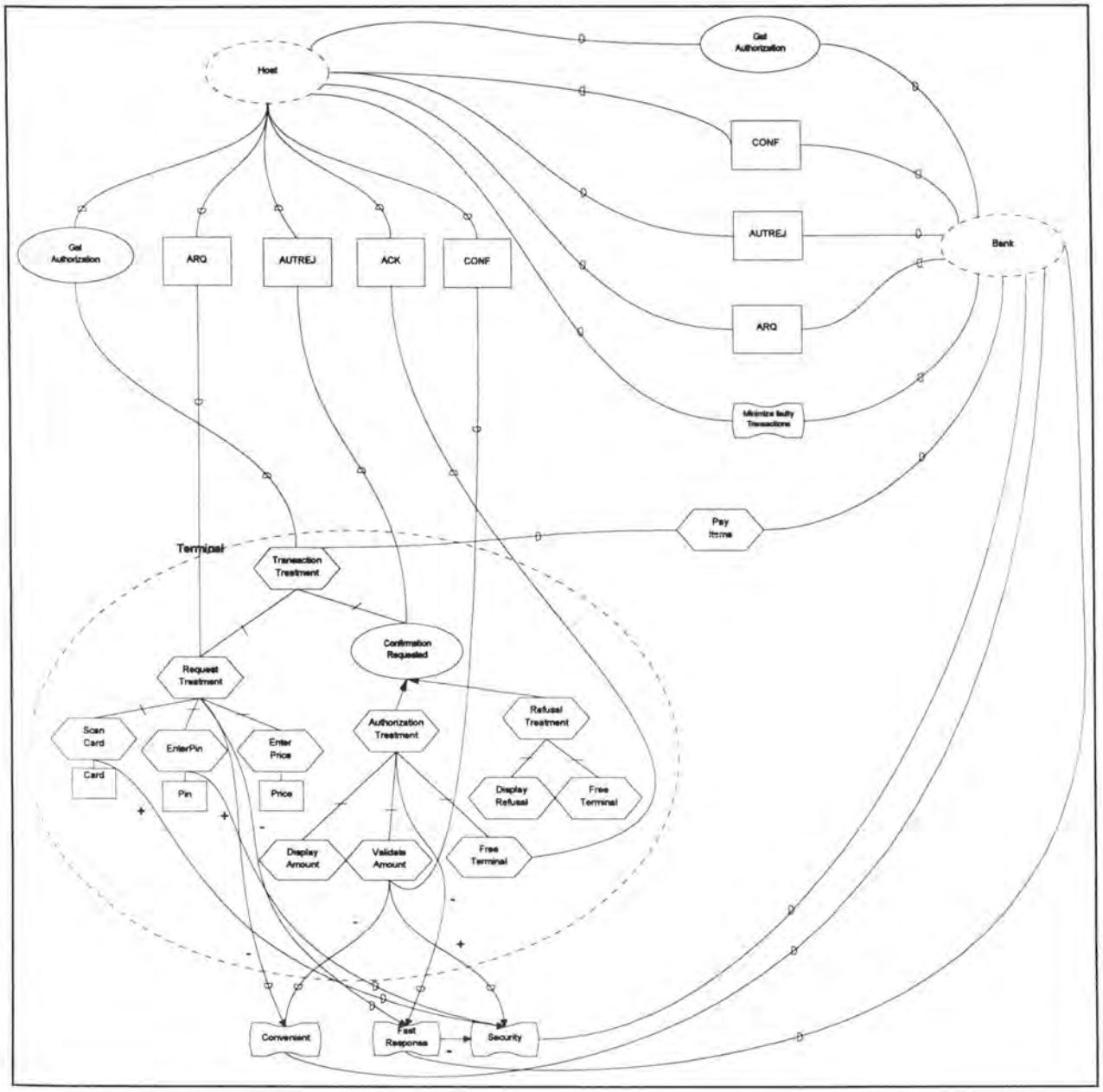


Figure 3.3. The Strategic Rationale model of the Terminal actor

The *RequestTratment* task allows the terminal to acquire the transaction information like the used debit card number (by executing the *ScanCard* task), the transaction amount (by executing the *EnterPrice* task) and the Pin code (by executing the *EnterPin* task).

To achieve the state represented by the *ConfirmationRequested* subgoal, the terminal needs the Authorization / Reject (AUTREJ) message from the Host. Depending on its content, the *AuthorizationTreatment* respectively the *RefusalTreatment* task is executed. In the case where the transaction request has been accepted, the *AuthorizationTreatment* task is executed which

consists in displaying and in validating the displayed transaction amount and in freeing the terminal.

The *ValidateAmount* task creates the Confirmation (CONF) message the Host depends on. The *FreeTerminal* task depends on the from the Host sent Acknowledge (ACK) message.

In the case where the transaction request has not been accepted, the *RefusalTreatment* task is executed. It consists in informing the customer that the requested transaction has not been accepted. The reason of the transaction's refusal is also displayed. The terminal is then freed and becomes available for future transaction processes.

b. The Host actor

After the Host has received the ARQ message from the terminal, the *TransactionTreatment* task is executed. The task consists in checking the validity of the received Pin code by executing the *Check Pin* task and by achieving the condition or state represented by the *Close Transaction* goal. Depending on the result of the made check, the condition or state represented by the goal component is achieved by executing the *AcceptedTreatment* or the *StoreInvalidTreatment* subtask.

Has a valid Pin code been received, the *AcceptedTreatment* subtask is executed which consists in executing the *ValidPinTreatment* task, the *AutrejTreatment* task and the *UpdateConf* task.

The aim of the *ValidPinTreatment* consists in finding out the account number belonging to the received debit card number (by executing the *FindOutAccNber* task) and in storing the received transaction information into the Host's database (by executing the *StoreInfo* task). The *ValidPinTreatment* task also produces the ARQ message the bank actor depends on.

The *AutrejTreatment* task depends on the bank's response (AUTREJ) message and consists in updating the previously stored transaction information. Once the information updated, a new AUTREJ message is created which represents the terminal's dependum.

The *UpdateConf* task updates the transaction information and depends on the confirmation (CONF) message it receives from the terminal. The task also creates a new confirmation (CONF) message which is forwarded to the bank.
In the case where an invalid Pin code has been received by the Host, the transaction information are stored (by executing the *StoreInvalidTreatment* task) and the Authorization /Reject (AUTREJ) message created. The message is used to inform the terminal that the requested transaction has not been accepted.

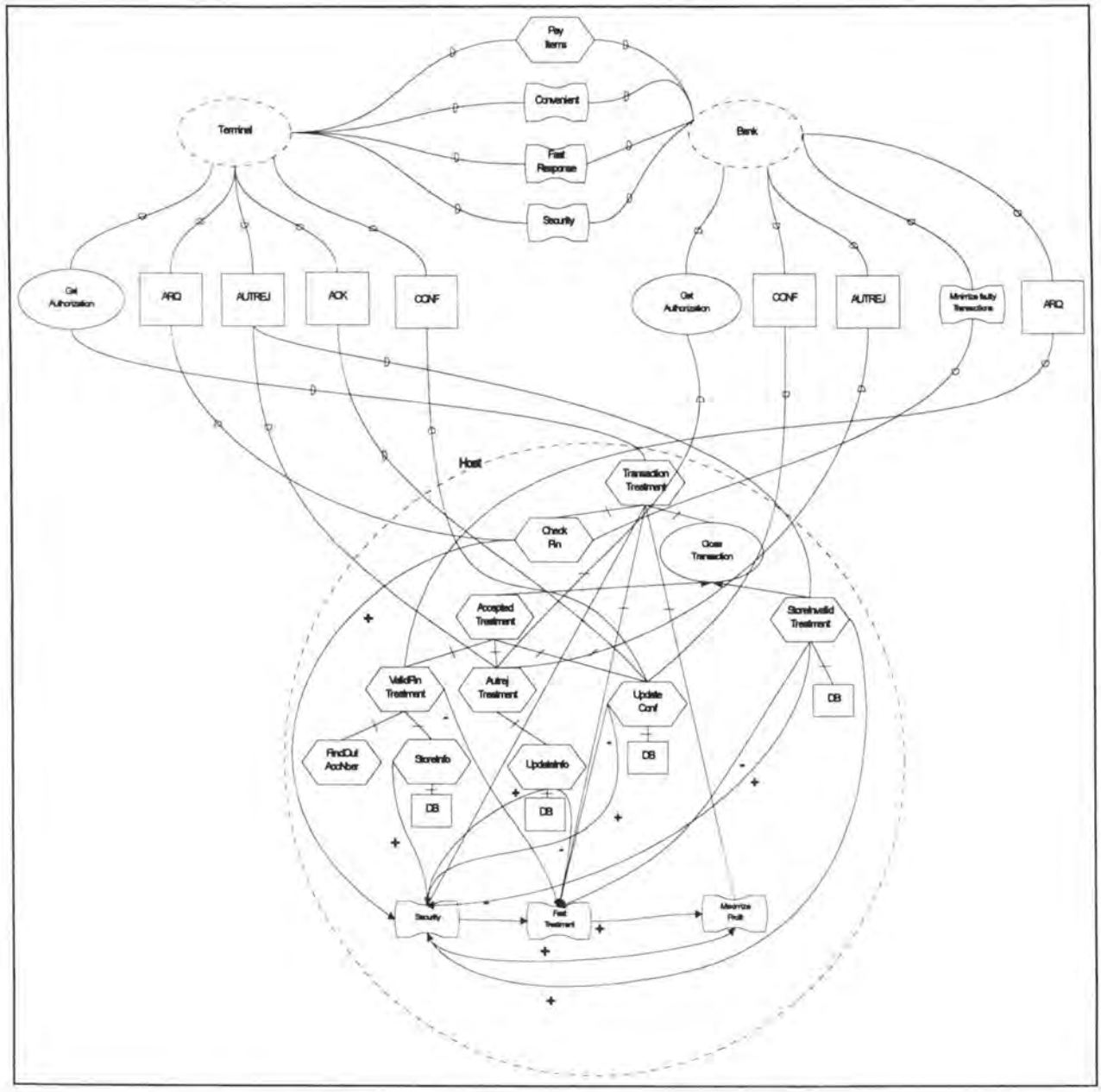


Figure 3.4. The Strategic Rationale model of the Host actor

By analyzing the depicted softgoals, we can find out that the Host is mainly interested in maximizing its profit and that two softgoals are contributing positively to this objective. A *Fast Treatment* allows the Host to increase the number of forwarded messages and at the same time its profit as we assume that there exist a direct link between the Host's profit and the number of executed transactions i.e. forwarded transaction messages.

A positively effect on the Host's profit has also the *Security* softgoal. The security softgoal represents the fact that the received and by-the-Host forwarded transaction information are stored and updated at the Host's level. The *Security* softgoal, however contributes negatively to the *FastTreatment* softgoal as storing and updating information takes time and slows down the transaction execution speed.

c. The Bank actor

In order to achieve the *Pay Items* dependency, the bank has to execute the *TransactionTreatment* task. The task itself is decomposed into two subtasks : (i) the *RequestTreatment* and (ii) the *ConfirmationTreatment* task.

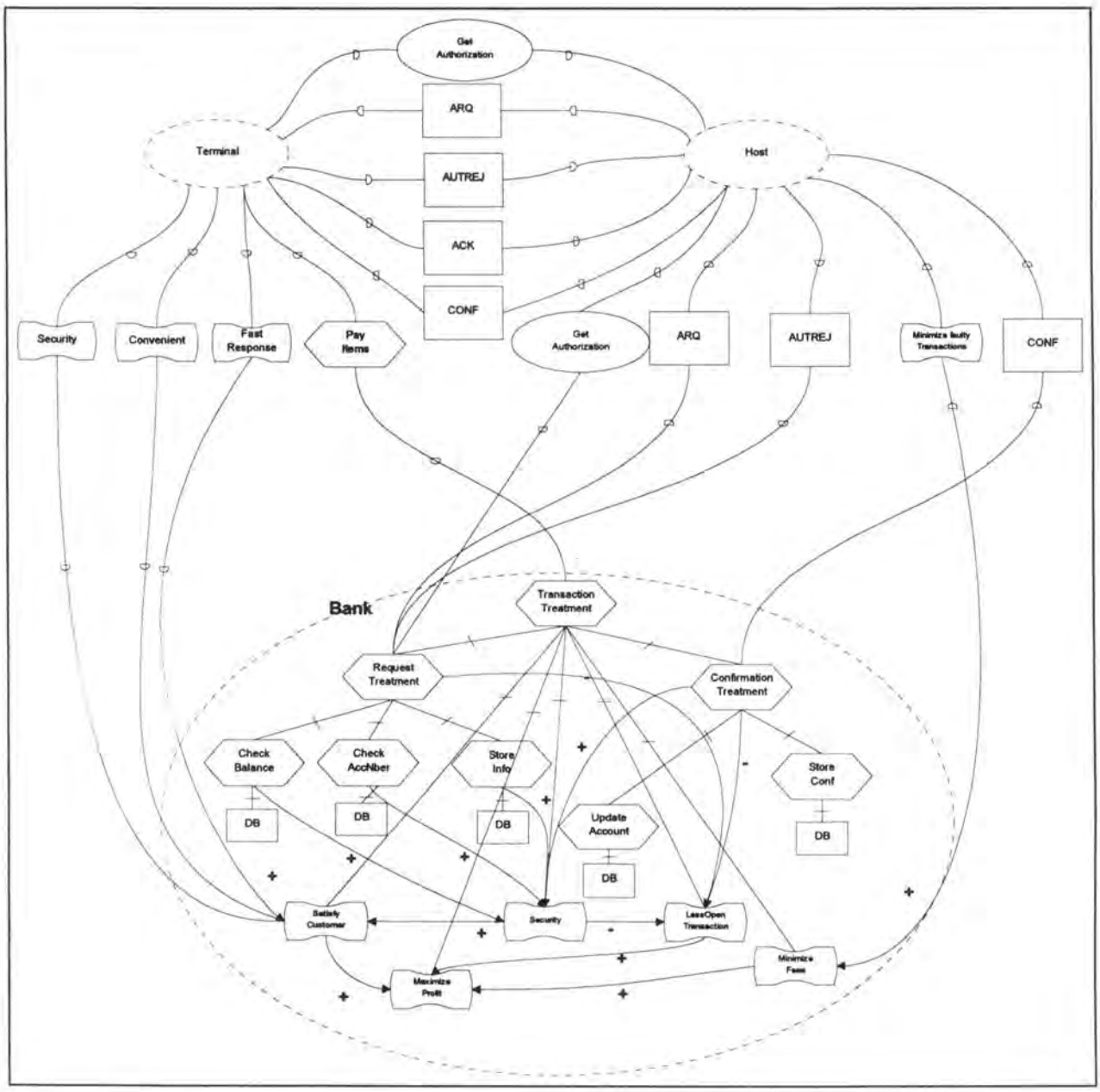


Figure 3.5. The Strategic Rationale model of the Bank actor

The *RequestTreatment* task is achieved by executing three subtasks : the *CheckBalance,* the *CheckAccNber* task and the *StoreInfo* task. The aim of the two first tasks consists in verifying the validity of the received account number respectively the customer's account balance. The *StoreInfo* task stores the received transaction information into the bank's local database.

The *ConfirmationTreatment* task depends on the Host's Confirmation message. Its execution consists in storing the terminal's confirmation into its database (by executing the *StoreConf* task) and in updating the customer's account balance (by executing the *UpdateAccount* task).

The bank's main objective is the maximization of its profit. Its profit is maximized by minimizing its fees as well as the number of *Open Transactions*. By *Open Transactions* we understand the transactions which have been authorized by the bank but whose confirmation the bank has not yet received.

We also assume that the *Satisfy customer* softgoal also contributes positively to the bank's profit as satisfied customer are more intended to spend money and use their card in order to pay for goods or services they purchase.

The storage of the transaction information as well as the made checks allow the bank to contribute positively to its *Security* objective and thus indirectly to the *Satisfy Customer* softgoal. Again, the fact of storing the transaction information slows down the transaction process as the storage takes time. We say that the storage contributes negatively to the *Less open transactions* softgoal.

Figure 3.5. depicts the Strategic Rationale model of the Bank actor.

After having described the Strategic Dependency as well as the Strategic Rationale model of our *Four Messages Protocol* example, we analyze now in the following sections the existing links between the i* and the Albert framework.

Section 2. The Task Dependency and decomposition

A task dependency describes a situation in which an actor (called the depender) depends on another actor (called the dependee) in order to get a certain task executed. A particularity of a task dependency consists in the fact that it is accompanied by a description on how the task should be executed. This description is represented in the Strategic Rationale model by using the task decomposition mechanism.

In our Credit Card Purchasing example, the terminal depends on the bank in order to pay the goods and services. As the terminal has to follow a certain predefined way of acting, the dependency is represented by a task dependency called '*Pay Items*'.

In order to find out which steps the dependee has to execute, let us analyze the Strategic Rationale model depicted by Figure 3.5.

In order to achieve the dependum, the bank has to execute the *TransactionTreatment* task. To execute the task as well as the related subtasks, the bank however needs certain information. These information are collected on the terminal's side by executing the *RequestTreatment* task depicted by Figure 3.3. The different obtained information are then sent in a message called Authorization Request (ARQ) message via the Host to the bank.

The i* example describes at design time a condition or state that has to be achieved. In the real world or let us rather say at run time, the i* condition or state is achieved by executing a set of

actions. And it is that particular set of actions as well as the related constraints which have to be specified at the Albert level.

As the execution of the actions described by the Albert specification allows us to achieve the same condition or state than the condition or state described at the i* level, we can say that it is possible to represent a task dependency and its decomposition by a set of Albert statements.

If we analyze for instance the *TransactionTreatment* task at the terminal's level depicted by Figure 3.3., we can find out that the task is decomposed into a subtask called *RequestTreatment* and a subgoal called *ConfirmationRequested*. The *RequestTreatment* task and all its subcomponents have to be executed in order to retrieve the transaction information the bank and the Host depend on.

If we associate now a substate to each of the subtasks and a state to the *RequestTreatment* task, the state associated to the *RequestTreatment* task is achieved by achieving the states associated to the different subtasks.

In order to find out, at the Albert level, the actions and the corresponding constraints which allow us to achieve the condition or state represented in the corresponding i* models, the task decomposition is very useful as it describes certain subconditions or substates that have to be achieved. Such a task decomposition may however be not complete.

As we have previously explained in Chapter 2, a Strategic Rationale model only represents components that are of a certain strategic interest for particular actor. Situations may however exist in which a certain subtask is not described by a task decomposition (as it has no strategic interest for an actor) but still has to be executed. The non existence of a certain strategic interest may be due to the fact that an actor may not take advantage from a given component i.e. that the component represents no opportunity for the actor. This may for instance be the case for a task that has to be executed by all of the different actors of a given organization or system and to which no dependency type can be attached to.

Such a task, even if it is not included in the task decomposition in the Strategic Rationale model, has to be performed by the actor. The job of the analyst consists in identifying these kind of tasks and include them in the Albert specification.

Figure 3.3 depicts that the terminal has to execute different tasks in order to collect the different transaction information. In practice, however, the simple fact of collecting the transaction information is not enough. The transaction information have to be sent to the Host by executing a certain action. Figure 3.3 however does not mention a task whose aim consists in forwarding the collected information to the Host. It only describes the existing resource dependency between the different actors and the way the dependum is achieved. The existing dependency however may be interpreted as an exchange of resources.
At the Albert level, the transfer of the transaction information from one agent to another has to be taken into account. Furthermore, in Chapter 1, we assumed that problems may occur during the exchange of the messages between the terminal and the Host actor. At the i* level, the Strategic Dependency respectively the Strategic Rationale Model inform us that there exist a resource dependency between the terminal and the Host actor but it does not mention or represent the fact that problems may occur during the transfer of the different information

between the terminal's and the Host's location. The only fact that may be represented at the i* level is related to the notion of degree of dependency.

In addition, not all i* tasks are or have to be translated into a set of actions and specified by an Albert statement. Figure 3.3. describes that the *RequestTreatment* task is decomposed into three subtasks. This means that 3 different subtasks have to be executed by the terminal before the main task *RequestTreatment* is achieved. At the Albert level, the analyst has to find out and specify a certain number of actions as well as the related constraints in order to achieve the *RequestTreatment* task. To do so, the analyst has to specify a certain number of actions that allow the achievement of the same situation than the situation represented by the different subtasks at the i* level. As the main task is decomposed into and achieved by executing the different subtasks and as the specified actions have been introduced in order to represent the different subtasks, the at-the-Albert level specified actions allow to achieve the *RequestTreatment* task.

The different actions and their constraints have however been introduced in order to represent the existing link between the actions and the subtasks. The main task i.e. the *Request Treatment* task has not been directly represented.

The Albert fragment associated to the terminal's *RequestTreatment* subtask is given by Figure 3.6. The *RequestTreatment* task and its decomposition is represented at the Albert level by the *Request* constraint of the Action Composition Constraints section.

The *Request* constraint specifies the sequential execution of five actions. Among those five actions, three actions have been introduced in order to represent the subtasks of the *RequestTreatment* task. These three actions are the *ScanCard*, the *EnterPrice* and the *EnterPin* action.

The *EnterTime* action has been introduced at the Albert's level in order to represent the fact that a terminal can only execute one single transaction at a given moment. The *SendRequest* action represents the exchange of the ARQ message between the terminal and the Host. The action is decomposed into two subactions *Forward_Ok* and *Forward_Ko* which are used in order to illustrate the fact that problems may occur during the exchange of the different messages.

We further associate a Precondition constraint to the *Request* action stipulating that the terminal has to be available before a new transaction process can be started i.e. the *Request* action be executed.

In our Albert specification, the terminal's status is represented by the boolean *Available* state component. We assume that the terminal is available i.e. is free and can be used for a new transaction process if the value of the state component is *true*. Otherwise, the terminal is regarded as non available.

The Initial Valuation constraints makes the terminal available at the beginning of its life-cycle. The terminal becomes unavailable for any further transaction process once the first action of the transaction process is executed. The fact that the terminal's status switches from available to unavailable is represented by the Effect of Action constraint.

BASIC CONSTRAINTS
DERIVED COMPONENTS
INITIAL VALUATION
Available := true
DECLARATIVE CONSTRAINTS
STATE BEHAVIOR
ACTION COMPOSITION

Request ↔ ScanCard(Card.arq) <> EnterTime(Date.arq, Time.arq) <> EnterPrice(Price.arq) <>
EnterPin(Pin.arq) <> SendRequest(arq)
SendRequest(arq) ↔ Forward_OK(arq) ⊕ Forward_KO(arq)

ACTION DURATION
OPERATIONAL CONSTRAINTS
PRECONDITION
Request : Available

EFFECTS OF ACTIONS
ScanCard(Card.arq) : []
Available := false
TRIGGERINGS
COOPERATION CONSTRAINTS

STATE PERCEPTION
ACTION PERCEPTION
STATE INFORMATION
ACTION INFORMATION
XK(SendMsgReq (arq, term_id).Host / true)

Figure 3.6. The Albert fragments associated to the terminal's *RequestTreatment* subtask

Finally, the Action Information constraint of the Cooperation Constraints section specifies the circumstances under which the Host perceives the execution of the *SendMsgReq* action. In our example, the terminal always informs the Host when it executes the action. As the messages exchanged between the terminal and the Host may be lost, we have to specify at the Host's level the fact that not all the-by-the terminal sent messages are perceived by the Host.

The different Albert statements represent the different i* components depicted by Figure 3.3. We can say that a tie exist between these statements as they have been introduced for the same purpose. This special tie may however be difficult to perceive in the final obtained Albert specification. In order to keep trace of the statements that have been introduced for the same purpose, a feature called *PostIt* has been introduced in Albert. A PostIt can be used in order to add comments to an Albert specification. A possible comment could consist in linking the statements that belong together i.e. that have been introduced for the same purpose

Section 3. The Goal Dependency and its decomposition

In opposition to a task dependency, a goal dependency does not specify how a state or condition has to be achieved. Is a goal component contained in a Strategic Rationale model, several alternative ways may exist which allow the achievement of a same condition or state (associated to the goal component). In a goal dependency, a certain freedom of acting is recognized to the actor.

This means that the actor is free to chose the way of proceeding that allows it to achieve the dependum in the case where several alternatives exist. The dependee may even delegate its responsibilities to another actor. Furthermore, the actor can even refuse to bring about the by-the-depender requested condition or state.

The goal dependency represents at the i* level a condition or state that the depender wants to be brought about. As we have mentioned in Chapter 2, the dependency link does not guarantee the depender that the dependum is or will be achieved or produced.

At the Albert level, the analyst has to find out a way that allows it to achieve the same condition or state than those expressed at the i* level. At the Albert level, the freedom of acting which existed at the i* level, does however no more exist. This is due to the fact that an Albert specification describes WHAT has to be done by the different agents. Has a special condition been realized or a particular action been executed, the specification forces an agent to act. The agent has no other choice.

Stated otherwise, we can say that whereas the notion of liberty of acting can be associated to a goal at the i* level, the same goal becomes a requirement which has to be realized by the different agents at the Albert level.

At the Albert's level, the analyst's job consists in specifying a set of constraints so that the at the i* described condition or state can be achieved. Similar to the task dependency link described in the previous section, the analyst's job can again be simplified by analyzing the goal decomposition as well as the existing means-ends links.

As explained in Chapter 2, different alternatives may exist in order to bring about the same dependum. The i* level recognizes an actor the liberty of choosing the way that suits him in the best way. Which alternative is finally chosen by the actor is however not specified. The evaluation of the alternatives i.e. their positive or negative contributions to the actors' softgoals allow us however to determinate in a certain degree the alternative that has the best chances to be chosen.

We can however not foreseen at hundred percent which alternative is finally chosen. The decision is taken by the actor and is mainly based on the alternatives' positive and negative contributions to the actor's softgoals. Other parameters like the actor's know-how and experience do also influence its choice.

As the notion of liberty of acting does not exist at the Albert level, a particular way of proceeding has to be chosen by the analyst (in the case where several alternative ways exist allowing to achieve the same condition or state). The final chosen way of acting may contain in alternatives which are represented by means-ends links.

Means-ends links allow us to represent the fact that different means allow a particular actor to achieve the same end. In our *Four Messages Protocol* example, the Host has to execute several checks after having received the transaction request (ARQ) message. Based on the received information, the Host decides whether the request has to be forwarded to the customer's bank or whether it is refused at its level (this is the case when the terminal has forwarded an invalid Pin code).

Figure 3.4 depicts that the *Close Transaction* goal component may be achieved by executing one of two existing means : the *AcceptedTreatment* task or the *StoreInvalidTreatment* task. Two alternative ways exist which allow to achieve the same goal. The *AcceptedTreatment* means-ends link is executed in the case the request has been accepted i.e. forwarded, otherwise the *StoreInvalid Treatment* means-ends link is executed.

In this particular case, both means have to be represented at the Albert level. The analyst, however, has to take care that it specifies clearly under which condition which mean has to be executed.

In our example, the Albert criteria can be expressed as follow : Has a valid Pin code been received by the Host, a certain number of actions, representing the i* *AcceptedTreatment* task, have to be executed. Otherwise, the analyst has to specify a set of actions and the related constraints which allow to represent the i* *StoreInvalidTreatment* task.

Once the different alternatives have been selected, the necessary means-ends links identified and the different criteria specified, the creation of the Albert specification becomes similar to the approach described in the previous section.

The Albert statements corresponding to the Host's means-ends link are given by Figure 3.7.

Figure 3.7. describes that after the Host has received the Authorization Request message from the terminal, the Host verifies the validity of the received debit card number as well as the validity of the Pin code. Both checks are made by executing the *DoCheck* action. Depending on the results of the two checks *CheckCard* and *CheckPin,* the *Valid_Request* respectively the *False_Request* action is executed.

Let us remark that both actions i.e. the *Valid_Request* and the *False_Request* action represent the upper part of the two means-ends links associated to the *Close Transaction* goal i.e. the *AcceptedTreatment* respectively the *StoreInvalidTreatment* tasks.

The subcomponents of the two means-ends links are represented at the Albert level by the *Valid_Request* respectively the *False_Request* decomposition.

The two Precondition constraints represent the criteria specifying when which mean-ends link has to be executed. They specify that, in the case where the *ForwardRequest* derived component (representing the result of the two checks) is true, the *Valid_Request* action has to be executed. Otherwise, the *False_Request* action has to be executed.

The combination of the Effect of Action constraints section and the Derived Constraints section allow us to assign the value *true* to the *ForwardRequest* state component in the case where a valid Pin code and debit card number have been received. Otherwise the value of the derived component is false.

Let us finally stress that the goal dependency and its decomposition plays an important part in the reengineering process. If an implemented system has to improved, its different goals represent potential points where improvements can be made as new alternatives can be associated to them and as the dependee is only interested in the achievement of the condition

BASIC CONSTRAINTS
DERIVED COMPONENTS
ForwardRequest = ValidCard ∧ PinMatches
INITIAL VALUATION
DECLARATIVE CONSTRAINTS
STATE BEHAVIOR
ACTION COMPOSITION
RequestTreatment ↔ Terminal.SendMsgReq(arq, term_id) <> DoCheckPin(Card.arq, Pin.arq)
DoCheck(Card.arq, Pin.arq) ↔ CheckCard(Card.arq) <> CheckPin(Card.arq, Pin.arq) <>
(Valid_ Request(arq, term_id) ⊕ False_ Request(arq, term_id))
Valid_ Request(arq, term_id) ↔ Search_AcNber(Card.arq, debit_nber) <>
StoreValidTrans(arq, term_id, host_ref) <>
CreateRequest(debit_nber, host_ref, request) <>
FindOutBank(debit_nber, bank_id) <>
Forward(request, bank_id)
False_ Request(arq, term_id) ↔ StoreInvalidTrans(arq, term_id, host_ref) <>
CreateReject(host_ref, rep) <>
SendMsgResponse(rep, term_id)
ACTION DURATION
OPERATIONAL CONSTRAINTS
PRECONDITIONS
Valid_Request(arq, term_id) : ForwardRequest
False_Request(arq, term_id) : ¬ ForwardRequest

EFFECTS OF ACTIONS

CheckCard(Card.arq) : []
ValidCard := In(Card.arq, Pin)
CheckPin(Card.arq, Pin.arq) : []
PinMatches := (Pin.arq = Pin[Card.arq])
TRIGGERINGS

COOPERATION CONSTRAINTS
ACTION PERCEPTION
STATE PERCEPTION
ACTION INFORMATION
STATE INFORMATION

Figure 3.7. The Albert statements corresponding to the Host's means-ends link

or state and not in the way how this is done.

Section 4. The Resource Dependency

In a Resource Dependency, an actor depends on another actor in order to get a certain resource. Used in a decomposition, it represents a physical or informational entity that an actor needs in order to achieve a certain state or condition.

The i* framework considers a resource as non-critical and the only problem that may occur is related to the fact that the resource is unavailable at a given moment. The non availability of the resource may have some serious consequences on the achievement or production of a certain dependum. The fact that a resource dependency exists between two actors does

however not mean that the resource is or will be produced or delivered. In the case where the problem is however perceived at time, a particular actor may take into consideration other existing alternatives in order to achieve or produce the same resource.

At the Albert level, a resource may be an internal or an external resource. In the case where the resource is an internal one, the resource is represented by a state component and is located inside the parallelogram of the actor. An external resource is also represented by a state component but is provided by an external agent.

A resource is created, used and modified by an action. In order to indicate which action uses which resource, the resource is indicated as a parameter of the action. The action's effect on the resource is specified by a statement of the Effect of Action Constraints section. The initial valuation of the resource, at its turn, can be specified by a statement of the Initial Valuation Constraints section. Finally, the resource's evolution can be restricted by a statement of the State Behavior Constraints section.

In the case where a resource is needed by an action in order to get executed, the unavailability of the resource implies the non execution of the action. As actions often produce an output which represents an input for other actions, those actions can also not be executed. A snowball effect can be the result which can theoretically lock the whole organization or system.

Such a deadlock is theoretically also possible at the i* level. It could however be prevented if alternatives exist which allow to bring about the same resource without using the unavailable one.

The conditions under which a certain external resource becomes available for a certain agent are specified by the statements of the Cooperation constraints.

In the case where an i* resource has to be represented in an Albert specification, the analyst has to start by specifying the resource's type. For a given agent, the analyst has then to find out whether the resource is an internal or an external one i.e. whether it belongs to the agent or whether it is provided by an external agent.

For each state component, the analyst may specify the state component's initial value and restrict its evolution if such a restriction exists. Is the value of the state component modified by an internal or external action, the action's effects have also to be specified.

In the case where the state component intervenes in the precondition of a particular action or in the case where its value triggers a particular action, the different circumstances have also to be described by using a constraint of the Precondition respectively the Triggering section.

Is a resource represented by an external state component or is a resource exported, the circumstances of the state perception or state information have also to be specified in the Cooperation Constraints section.

Let us remark that from a theoretical viewpoint, an i* resource can be easily represented at the Albert level. In practice, however, some problems may occur.

First of all, a resource does not simply represent a simple flow at the i* level but can be accompanied by another dependency type. The aim of the analyst consists, at the Albert level, in identifying the different dependencies which are related to the resource dependency and represent both the resource and the different related dependency types in the Albert specification.

The Strategic Dependency model, for instance, depicted by Figure 3.2. describes the fact that several resource dependencies exist between the Host and the Bank actor. The Bank actor for example depends on the Authorization Request (ARQ) and the Confirmation (CONF) message. In both cases, the dependee is the Host agent. In order to represent our *Four Messages Protocol* example at the Albert level, the analyst has to find out, among others, the resources' types and the way and circumstances they are produced. Once the characteristics of the resources determined, the analyst can theoretically start with the specification of the resource at the Albert level.

As a resource does not represent a simple flow at the i* level, the resource can be influenced or let us rather say accompanied by other dependencies. In our case, the *Minimize Faulty Transactions* softgoal influences the exchanged resources. In fact, the softgoal dependency requires a certain way of acting that allows the bank to minimize the number of its faulty transaction. Such a way can however have a serious impact on the resources' characteristics.

In addition, minimizing the number of faulty transactions can also have a serious impact on the way and circumstances the resource is produced so that the analyst's job is not as simple as first expected.

The second problem that may occur while representing a resource component consists in the fact that an i* resource is not always represented by an Albert state component. In certain cases, it is of a certain interest to represent a resource or let us rather say an exchange of a resource by an action.

This is for instance the case in our *Four Messages Protocol* example. If we focus on the Authorization Request (ARQ) message exchanged between the Host and the bank, we could represent the ARQ message by a state component and specify the circumstances under which it is perceived by the dependee.

As those circumstances are based on expressions and as those expressions contain state components, the customer's bank has to evaluate those expressions i.e. it has to evaluate a certain number of state components in order to find out whether it perceives or not the state component(s) representing the ARQ message.

From a theoretically viewpoint this is possible but as in practice important distances may exist between the Host's and the bank's location, this sort of mechanism becomes in practice very difficult and even impossible to realize.

A solution to this problem consists in representing the exchange of information by an action where the parameters of the action represent the different exchanged information. In our case, the Bank perceives the fact that the Host is sending the transaction information by executing a certain action and perceives the exchanged information by analyzing the action's parameters.

The perception of the action can be used in order to start the execution of other actions at the bank's level (by using a constraints of the Action Composition Constraints section) and the action's parameters can be used as input for other actions.

In order to illustrate an existing resource link between the i* and the Albert framework, let us analyze, for instance, the ARQ resource dependency between the Host and the bank. As previously mentioned, the bank actor depends on the ARQ message in order to take a decision whether the requested transaction can be accepted or not.

At the Albert level, the analyst has to specify the resource's type, the fact whether the resource is represented by an internal or external state component, the way how the resource is produced, used and forwarded.

Based on our *Four Messages* Protocol example, the Host receives the transaction request from the terminal. The contents of the received message is described by the ARQ type. The received ARQ message contains : (i) the date and time when the transaction has been requested, (ii) the used debit card number, (iii) the transaction amount and (iv) the Pin code.

After the Host has executed the-at-the Albert level specified actions, a message of type REQUEST is forwarded to the customer's bank. The difference with the received ARQ message consists in the fact that in the forwarded message, the number of the debit card has been replaced with the customer's account number.

The types of the two messages are given by :

```
ARQ = CP (Date : DATE ; Time : TIME ; Card : CARD ; Price : INTEGER ; Pin : PIN )

REQUEST = CP (Debit_nber : DEBIT, Price : PRICE ; Host_ref : REF_H, Date : DATE,
              Time : TIME)
```

As previously mentioned, the exchange of the different message is represented by the execution of an action. In our example, the request message is sent to the customer's bank by executing the *Forward* action.

The *Forward* action is decomposed by using a constraint of the Action Composition Constraints section. The corresponding subactions are the *ForwardOk* and the *ForwardKo* action. Both actions are used in order to represent the fact that problems may occur at the destination's location i.e. at the bank's level.

$$Forward(request, bank\_id) \leftrightarrow (ForwardOk(request, bank\_id) \oplus ForwardKo(request, bank\_id))$$

The assumption taken in the first chapter related to the quality of equipment and lines used by and between the Host and the bank is still valid. Let us remind that we assumed that messages may only be lost during the exchange between the terminal and the Host. Some internal problems (at the bank's level) may however imply that the bank's response does not arrive at time at the Host's location. In this case, a Timeout occurs and the transaction process is aborted by the Host.

The fact that a Timeout may occur is represented by using two Action Composition constraints :

ForwardOk(request, bank_id) ↔ SendRequest(request, bank_id) <> ResponseTreatment
ForwardKo(request, bank_id) ↔ SendRequest(request, bank_id) <> TimeOut(Host_ref.request)

The constraints describe that the *ForwardOk* action is executed in the case where the *SendRequest* action is followed by the reception of the bank's response within a certain interval of time. The *ForwardKo* action is executed in the case where the Timeout occurs.

The interval of time during which the Host waits in order to get the response is represented by an Action Duration constraint :

ACTION DURATION
|ForwardOk(request, bank_id) | ≤ TimeOutPeriod

The occurrence of a Timeout is expressed by the following constraint :

|ForwardKo(request, bank_id) |> TimeOutPeriod

As the action of sending messages from the Host to the bank is used in order to trigger certain actions at the bank's level, the bank actor has to be informed of the execution of the *SendRequest* action.

COOPERATION CONSTRAINTS
ACTION INFORMATION
XK(SendRequest(request, bank_id).Bank / true)

Section 5. The Softgoal Dependency and their contribution

A softgoal describes a non-sharpened condition that the depender wants to be achieved but which may be subject to interpretations. The advantage of a softgoal consists in the fact that it may help the analyst to chose an appropriate way of acting by giving it some additional information about the actor's desires and wants.

At the specification level, the notion of i* softgoal can be linked to the notion of *Non Functional Requirements* (NFR). As an Albert specification only specifies the Functional Requirements of a given application or system, no direct link can be made between an i* softgoal and an Albert component.

At the i* level, softgoals play an important part in the decision process of an actor. In the case where several alternative ways exist which allow the actor to achieve the same condition or state, the softgoals can be used in order to evaluate each of the existing alternatives so that a final choice can be made by the actor. In other words, softgoals are used to foresee at a certain degree the behavior of a given actor. The actor is however free and may even chose an non logical alternative.

At the Albert level, the notion of liberty of acting, the possibility to make choices and the notion of softgoals do not exist. An Albert specification describes what has to be done by a particular agent once certain conditions or states are achieved. The agent has no choice i.e. is forced in a certain way to act. If alternatives exist, the analyst has to specify clearly the condition under which a certain alternative has to be executed.

The non existence of the i* softgoal notion at the Albert level can however be moderated by two facts. First, the aim of a specification consists in describing in a particular way the desires and needs of a given application or system. A good specification describes what is expected without any ambiguities. The obtained specification however never describes all the customer's requests, wants and desires. By using Functional Requirements, there exist always a certain part that can not be entirely represented. Those facts are however not eliminated from the specification process but are expressed by using Non Functional requirements. A usual way consists in expressing them in a textual way.

Second, we can say that an Albert specification does not *explicitly* contain the notion of an i* softgoal. The facts described at the i* level by the softgoals are however implicitly contained in an Albert specification.

If, at the end of a specification process, the analyst is asked why he or she has chosen a particular way of acting, for instance why he or she has introduced the Timeout notion at the Albert level, the analyst may reply for instance that the organization's interest consisted in getting a '*Fast Transaction Response*' and that the Timeout mechanism allows to intervene once a deadlock occurred.

Stated otherwise, we can say that even if the Albert language does not contain a notion similar to the i* softgoal, the at the i* represented softgoals are contained implicitly in the corresponding Albert specification. This under the assumption that the analyst has respected the at the i* level expressed wants and desires.

Section 6. The Liberty factor of an actor respectively an agent

A dependency exists in a situation in which a depender depends on a dependee in order to get a certain dependum achieved or realized. The subject of the dependency may be a task, a resource or a softgoal. An existing dependency does however not ensure the depender that the dependum is realized. The depender may not have the time or may even not be interested in bringing about the dependum.

The failure can have some implications for the depender and the long term relationship between the dependee and the depender can also be affected. In order to evaluate those implications, the notion of degree of dependency has been introduced in the i* framework.

In the case where the depender perceives at time that the dependum is or will not be brought about, the depender has, in certain situations, the possibility to take into account existing alternatives which allow him or her to get the requested dependum.

As mentioned several times before, the liberty of acting which exists at the i* level does no more exist at the Albert level. The aim of an Albert specification consists in describing how a

certain condition or state may or has to be achieved by specifying a certain set of actions as well as the related constraints. Is a certain condition realized and the execution of a particular action linked to that condition (by using a Triggering constraint), the Albert agent has to execute that particular action.

The liberty of acting does no more exist at the Albert level but this fact does not have a negative implication on the Albert framework. An Albert agent plays simply a more passive role than an actor at the i* level but thus is responsible for the actions that have to be executed by it. An Albert agent is simply following a given specification by the letter, and behaves like requested.

Conclusion

Nowadays, it becomes more and more difficult to develop and implement complex systems. Difficulties also appear when it comes to improve such an existing system. Most of the problems are mainly based on the fact that the documents that have been used in order to develop and implement the system do not contain detailed information which allow the analyst to improve the given system.

The limitation of those documents mainly consists in the fact that they only describe WHAT has to be realised by the different entities and not WHY the entities have to act in a certain predefined way. The Why is however needed when the analyst has to rewrite and improve a given organisation or system as it provides important information about the organisation or system as well as the context in which the organisation or system is embedded.

In Chapter 1, we have presented a formal requirements specification language called the Albert II language allowing to describe what has to be realised by the different entities of a given system. The Albert II language has been developed at the Facultés Notre Dame de la Paix in Namur, Belgium and suits particularly for representing real-time distributed and cooperative systems.

In Chapter 2, we have described the i* framework developed at the University of Toronto, Ontario, Canada. The i* framework allows to represent the WHYs of a given organisation or system by taking an agent orientated approach. The i* framework allows to represent a given organisation or system by describing the existing agents, the existing dependencies between them and the agents' intentional and strategical behaviour in order to bring about a certain source of dependency, called dependum at the i* level.

Whereas the Albert II language, presented in Chapter 1, only allows to describe a particular way of acting, the i* framework also allows to represent different alternatives way which allow a particular agent to bring about a certain dependum. In addition, the framework allows to represent the opportunities and vulnerabilities i.e. the positive and negative contributions of a given alternative way from an agent's viewpoint.

In Chapter 3, we have described why it is of a certain interest to combine the Albert II and the i* framework. Even if the two frameworks are used at different levels or lets rather say at different moments in the specification process, the analyst can take advantage of the combination of the two frameworks.

First of all, the use of the i* and the Albert II framework allows the analyst to represent both the *WHAT*s and the *WHY*s of a given organisation or system. At the Albert level, the use of the i* framework allows the analyst to facilitate its preliminary work which consists in analysing and understanding a given system as well as the context in which the system is embedded. The i* models also describes the different existing alternatives (in the case where alternatives exist) so that the analyst's work becomes much more easier.

The Albert II language, at its turn, may be used in order to specify in a more detailed way the behaviour of the different agents by specifying for instance the constraints that have to be

respected by the different agents. The Albert language can also be used in order to find out whether the at the i* represented system may be implemented or not by specifying and verifying a certain number of constraints related to the system.

In the case where problems are detected at the Albert level, the analyst may go back to the i* level and chose or search another existing alternative. The going back and forth may be repeated until a valid specification is finally obtained.

Chapter 3 also describes the existing links between the Albert II and the i* framework. The fact that a link exist between both frameworks does however not mean that a given i* model may be easily translated or lets rather say represented at the Albert level by applying a certain number of predefined rules. The analyst still needs a certain knowledge about the system that has to be realised or improved as well as the context in which the system is or has to be embedded.

References

[1] Eric Dubois, Philippe Du Bois and Jean-Marc Zeippen. *A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems*, Institut d'Informatique, Facultés Universitaires de Namur, (Belgium)

[2] Philippe Du Bois. *The Albert II Language, On the Design and the Use of a Formal Specification Language for Requirements Analysis*, PhD thesis, Facultés Universitaires Notre Dame de la Paix, Namur (Belgium)

[3] Bernard Jungen. *The Albert II Reference Manual, Version 1.1*, CAT project, 1996

[4] Eric Dubois and Michaël Petit. *A Formal Requirements Engineering Framework for CIM Infrastructures Reengineering*, Computer Science Department, University of Namur.

[5] Eric Yu. *Modelling Strategic Relationships For Process Reengineering*, PhD thesis, University of Toronto, Ontario (Canada)

[6] Ian Sommerville, *Software Engineering*, Addison Wesley, 1996

[7] Ross & Schoman, *Structured Analysis for Requirements Definition* in Classics in Software Engineering, Edited by Edward Nash Yourdon, Yourdon Press

Appendix : The Two Messages Protocol example

Section 1. Description of the Two Messages Protocol example

In the *Two Messages Protocol* example, two messages have to be exchanged between the different involved agents before the transaction process is completed.

The transaction process begins by sending an Authorisation Request (ARQ) message to the C-ZAM Host. The structure of the sent ARQ message is similar to the structure of the ARQ message in the *Four Messages Protocol* example.

Once the ARQ message has been sent to the C-ZAM Host, the terminal waits a certain period in order to get a response from the Host. If the Host's answer does not arrive within that period, a Timeout occurs.

Having received an Authorization Request (ARQ) message from the terminal, the C-ZAM Host verifies if the customer has entered a valid PIN code. Is this the case, the C-ZAM host saves the transaction information into its local database, determinates the customer's bank to which the message has to be forwarded, replaces the customer's debit card number by the customer's account number and forwards the modified transaction request (ARQ) to the bank. The Host then waits a certain time in order to get the bank's response. If the bank's response does not arrive within that period, a Timeout occurs and the transaction process is aborted.

If the customer has entered an invalid PIN code, the transaction request is refused at the Host's level and is not forwarded to the customer's bank.

On the basis of the received transaction information (i.e. the customer's account number and the transaction amount), the bank decides whether the transaction request can be accepted or not. A transaction request is accepted if the customer has enough money on its account in order to cover the transaction. In order to find out whether the transaction is covered or not, the same procedures than in the *Four Messages Protocol* example are applied.

After the transaction information and the bank's decision have been recorded, the customer's account balance is updated (in the case where the transaction request has been accepted) and the bank's response, called Authorization / Reject (AUTREJ) message, is sent to the C-ZAM Host.

After the reception of the bank's response, the Host updates its corresponding transaction record based on the received information and transfers the received message to the terminal from where the transaction request has been emitted.

If a message arrives at the terminal informing the customer that the transaction request has been refused, the reason of the transaction's refusal is displayed on the terminal's LCD screen. In the case where the request has been accepted, the customer is informed and the transaction process is finished. In opposition to the *Four Messages Protocol* example, the customer does

not need to confirm the transaction amount. As soon as the bank's response arrives and has been displayed at the terminal's location, the transaction process is finished.

The occurrence of a Timeout at the terminal's level

In the case where a Timeout occured i.e. in the case where the answer of the customer's bank (respectively the Host) is not received by the terminal from the C-ZAM Host after a certain period of time, the transaction is aborted and a message, called Authorization Request Canceling (ARQ-) message, is sent to the customer's bank informing it that the previously sent Authorization Request (ARQ) message has to be cancelled.

To ensure that the ARQ- message arrives at the Host's location, a mechanism similar to the one described in the 4MP example is used. We assume that the information contained in the ARQ- message are stored in the terminal's internal memory and that the ARQ- message has to be sent as long as the terminal does not get an Acknowledge Canceling (AOK) message in return. It is only after the terminal receives a valid AOK message that the corresponding ARQ- message is removed from its memory.

When a ARQ- message arrives at the Host's level, the Hosts checks if it has already forwarded the received ARQ- message to the customer's bank. Is this not the case, the ARQ- message is forwarded to the customer's bank and an AOK message is returned to the terminal. In the case where the received ARQ- message has already been forwarded, an AOK message is sent to the terminal and the message is not forwarded to the customer's bank.

The customer's bank evaluates the received ARQ- message, stores the transaction information into its local database and updates the customer's account balance (i.e. adds the transaction amount to the customer's account balance in the case where the transaction request has previously been accepted i.e. in the case where the transaction amount has previously been retrieved from the customer's account).

Special case : An ARQ- message arrives before its corresponding ARQ message

An ARQ- message can theoretically arrive before its corresponding ARQ message. This is for instance the case when a ARQ message is jammed or lost whereas its corresponding ARQ- message arrives at destination without any troubles.

At the Host's level, the ARQ- message is stored and forwarded to the customer's bank. If the corresponding ARQ message arrives, it is stored and forwarded to the customer's bank.

At the bank's level, the ARQ- message is stored in the bank's database. If its corresponding ARQ message arrives afterwards, it is stored in the database but is not evaluated.

The exchanged messages are represented by Figure 5.1a. and 5.1b. Figure 5.1a. describes the exchange of the two messages. The Authorization Request (ARQ) message contains the terminal's transaction request. The Authorization /Reject (AUTREJ) message encloses the bank's response (as well as the reason of the refusal in the case where the request has not been accepted).

Figure 5.1b. describes the different exchanged messages in a transaction request in the case where a Timeout occurs at the terminal's level.

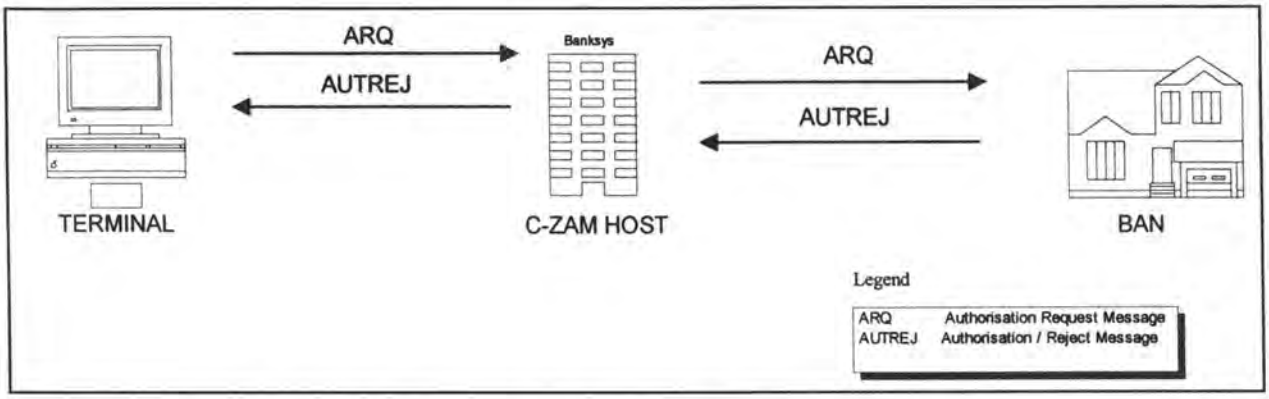


Figure 5.1a. The regular exchanged messages in our 2 Messages Protocol example

After the terminal has waited a certain period of time in order to get the bank's response, a Timeout occurs and an Authorization Request Canceling (ARQ-) message is sent to the bank via the Host. The terminal then waits in order to get the Canceling Acknowledge (AOK) message from the Host.

In the example depicted by Figure 5.1b., we assume that the first sent ARQ- message does not arrive at the destination or that the Host is not able to sent an AOK message in return. It's only after the terminal has sent the message for the second time that the corresponding AOK message arrives at the terminal's location.

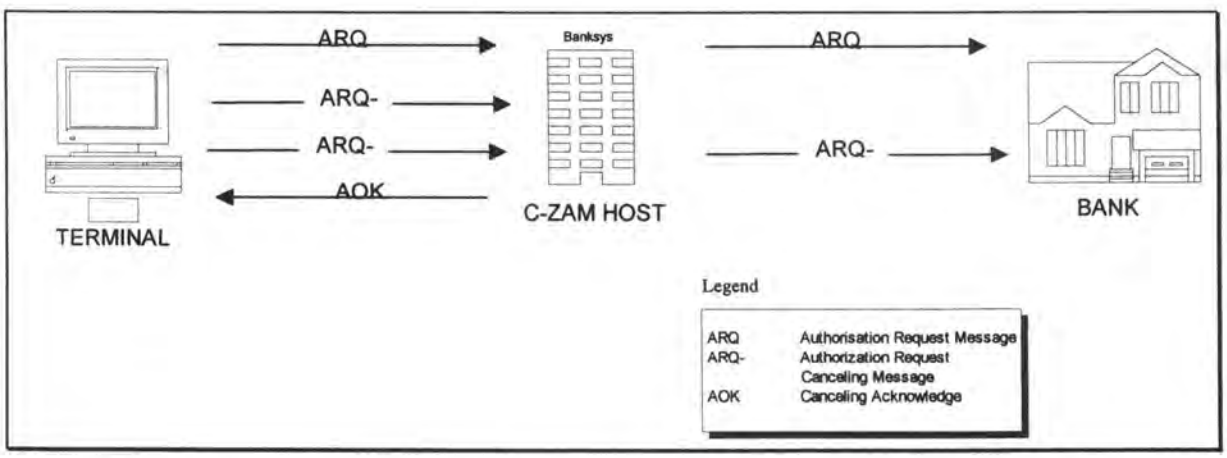


Figure 5.1b. The occurrence of a Timout in our 2 Messages Protocol example

The content of the different messages is depicted by Figure 5.2a. respectively Figure 5.2b.

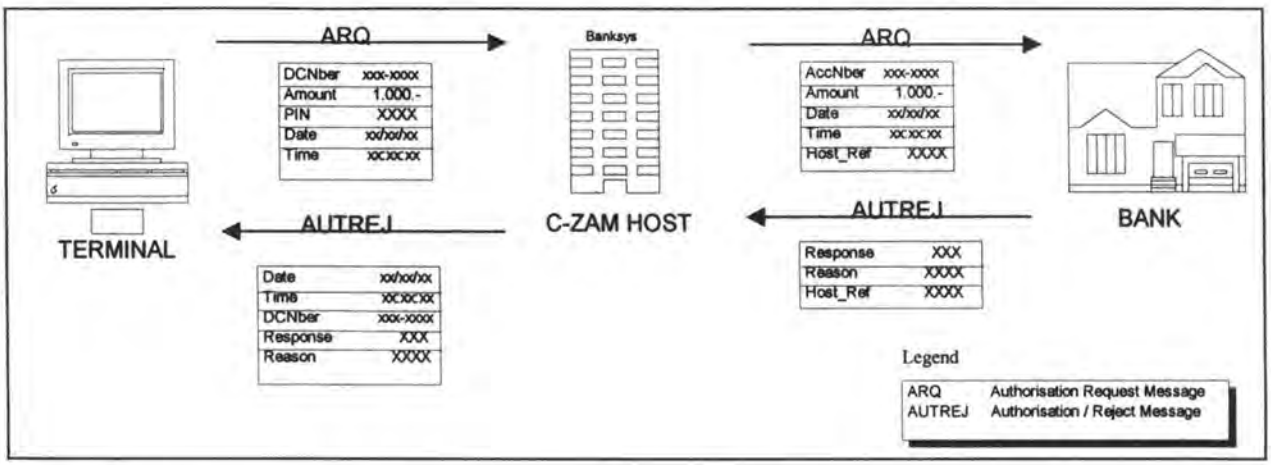


Figure 5.2a. The contents of the different exchanged messages in the 2 Messages Protocol example

In a regular transaction process (i.e. in a process where the request is not cancelled), the same information than those in our 4MP example are exchanged.

In the case where a Timeout occurs, we assume that the Host forwards the ARQ- message to the bank and that the message contains the same information than those contained in the ARQ request message. The content of the ARQ- and AOK messages is given by Figure 5.2b.

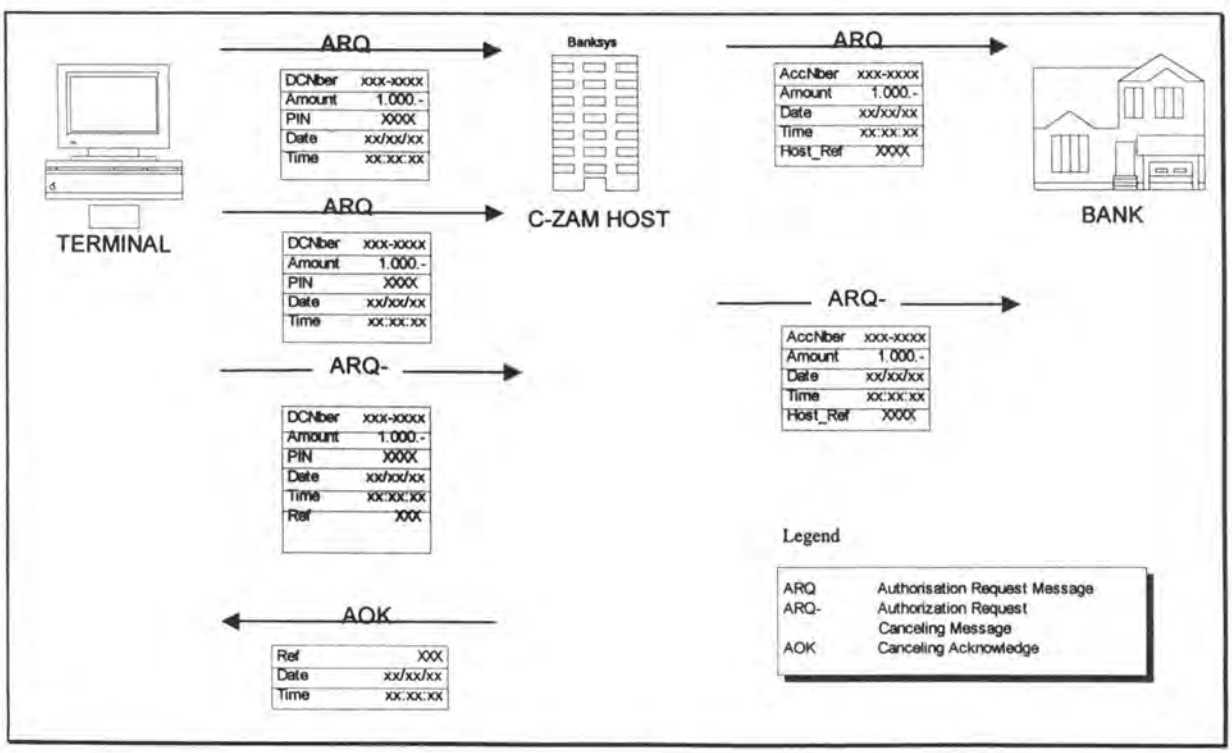


Figure 5.2b. The occurrence of a Timeout in the 2 Messages Protocol example

Section 2. The i* models of the Two Messages Protocol example

1. The Strategic Dependency model of the 2MP example

The Strategic Dependency model of our 2MP example depicted by Figure 5.3., contains, similar to our *Four Messages* Protocol example, 3 actors : the Terminal, the Host and the Bank actor. The difference between both example consists in the fact that in the 2MP example only 2 messages have to be exchanged (instead of four like in the 4MP example).

In the case where a problem occurs in the 2MP example during the transaction process, the transaction process is aborted and two additional messages have to be exchanged in order to ensure that the transaction is actually canceled.

The actors' dependencies are the same than in the *Four Messages Protocol* example with the exception that the ARQ- resource dependency represents the Authorization Request Canceling message and the AOK resource dependency the Acknowledge Canceling message.

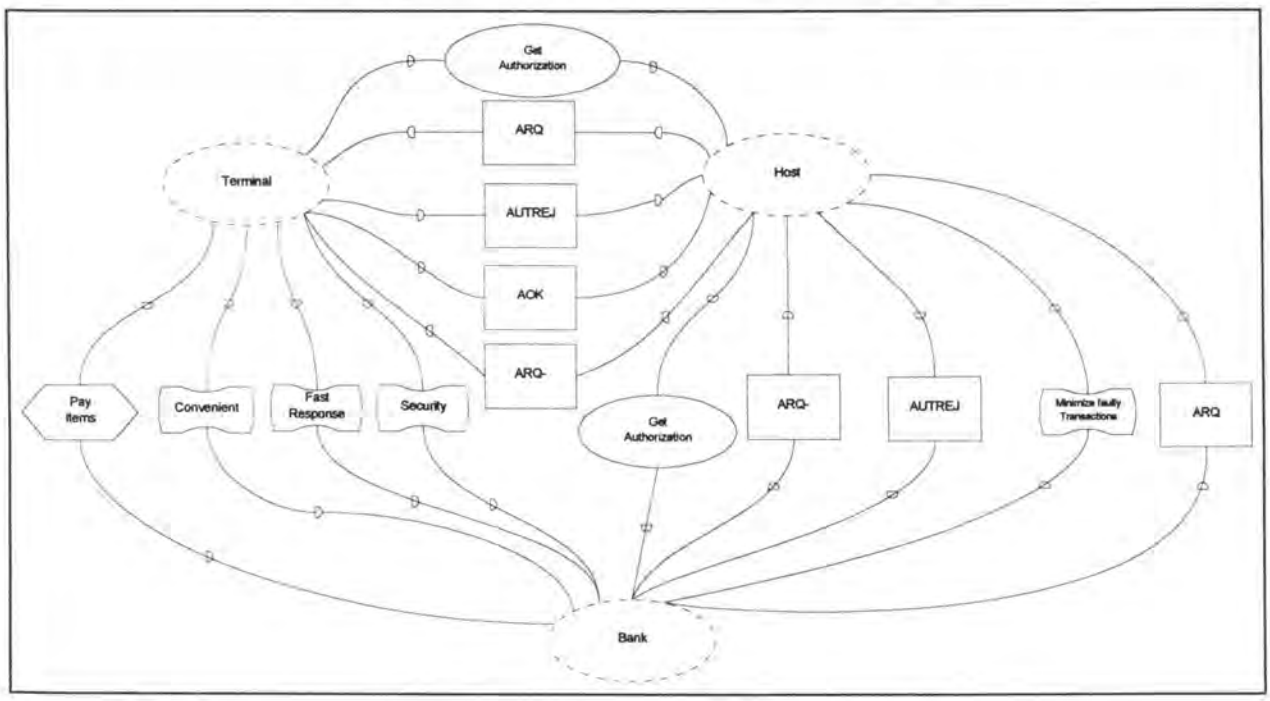


Figure 5.3. The Strategic Dependency model of the 2MP example

2. The Strategic Rationale model of the 2MP example

a. the Terminal actor

Figure 5.4. depicts the Strategic Rationale model corresponding to the terminal actor in our *Two Messages Protocol* example.

The Strategic Rationale model of the terminal actor describes that in order to execute the transaction process, the terminal has to execute the *Transaction Treatment* task. The task is decomposed into a task called *RequestTreatment* and a goal called *Timeout*.

The *RequestTreatment* task is decomposed into three subcomponents and allows to collect the necessary transaction information.

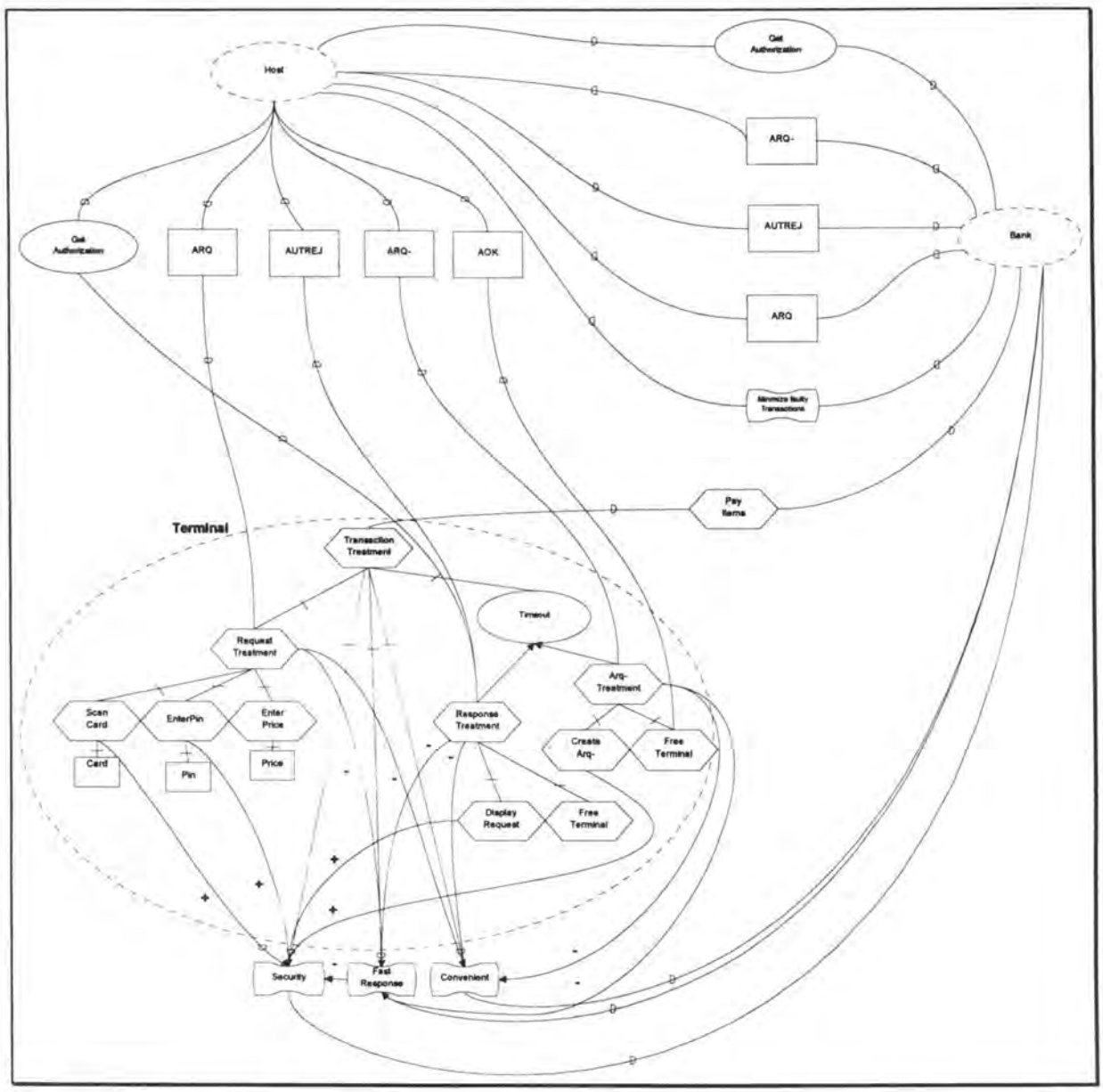


Figure 5.4. The Strategic Rationale model of the Terminal actor

The *Timeout* goal can be achieved by executing one of the two means-ends links called *ResponseTreatment* respectively *Arq-Treatment* and represents the fact that the sending of the ARQ message is followed by the reception of the AUTREJ message or the occurrence of a Timeout.

In the case where a Timeout occurs, the *Arq-Treatment* means has to be executed ; otherwise the *ResponseTreatment* means.

The *ResponseTreatment* means consits in displaying the received AUTREJ message and in freeing the terminal. The *Arq-Treatment* means consists in creating the ARQ- message the Host depends on. After the reception of the AOK message, the terminal becomes again available after the execution of the *FreeTerminal* task.

b. the Host actor

Figure 5.5. depicts that depending on the received message, the Host executes the *TransactionTreatment* respectively the *TransactionCanceled* subtask. The *TransactionTreatment* task is executed after the reception of an Authorization Request (ARQ) message; the *TransactionCanceled* task after the arrival of an Authorization Request Canceling (ARQ-) message.

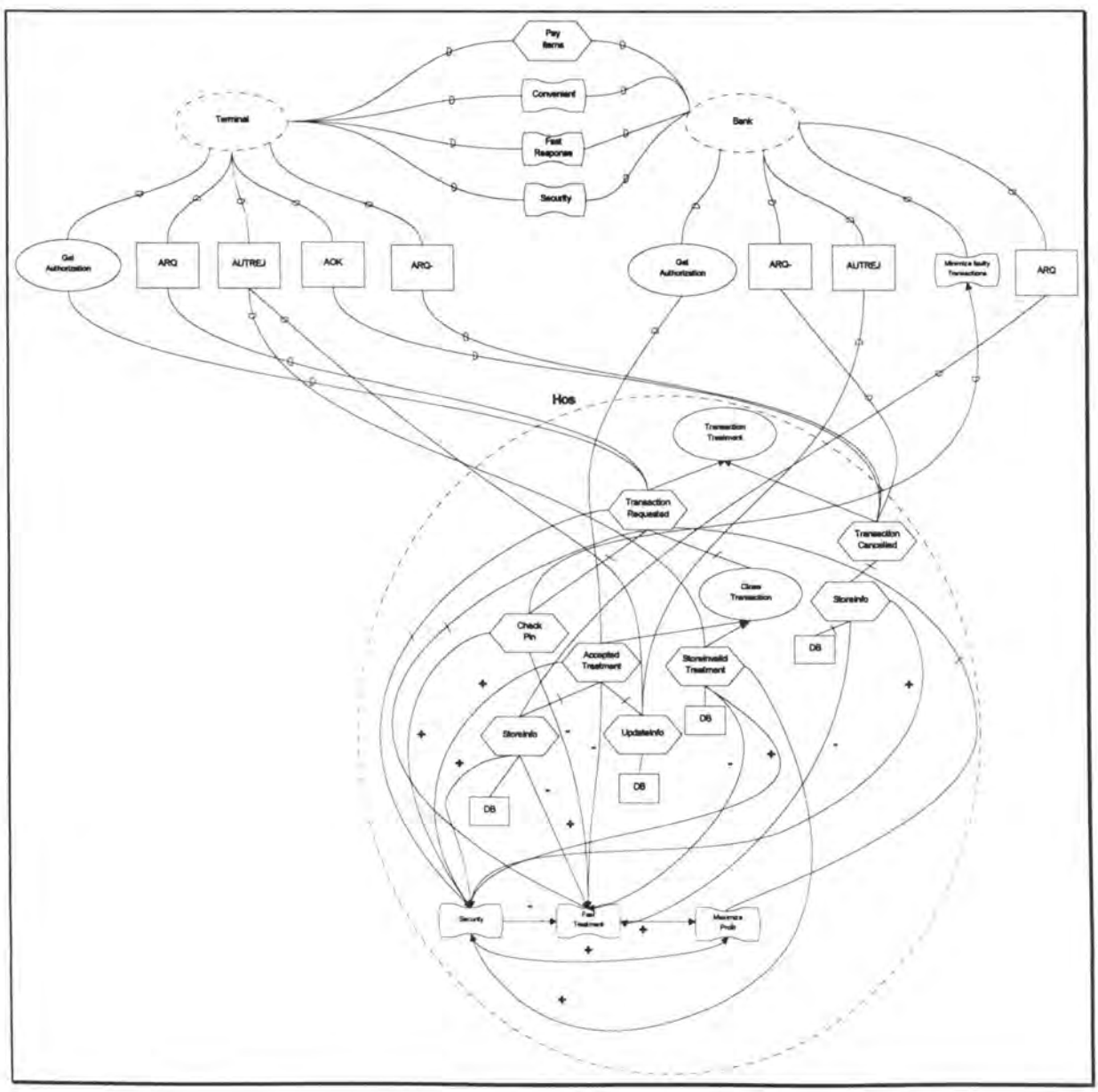


Figure 5.5. The Strategic Rationale model of the Host actor

The*TransactionCanceled* task stores the received information into the Host's database and forwards the received ARQ- message to the bank. Finally, a Canceling Acknowledge (AOK) message is sent to the terminal.

c. the Bank actor

The Strategic Rationale model describing the bank's behavior is depicted by Figure 5.6.

The participation of the bank actor in the transaction process consists in achieving *TransactionTreament* goal. To do this, the bank has to execute one of the two means-ends links called *RequestTreatment* respectively *TransactionCancel.*

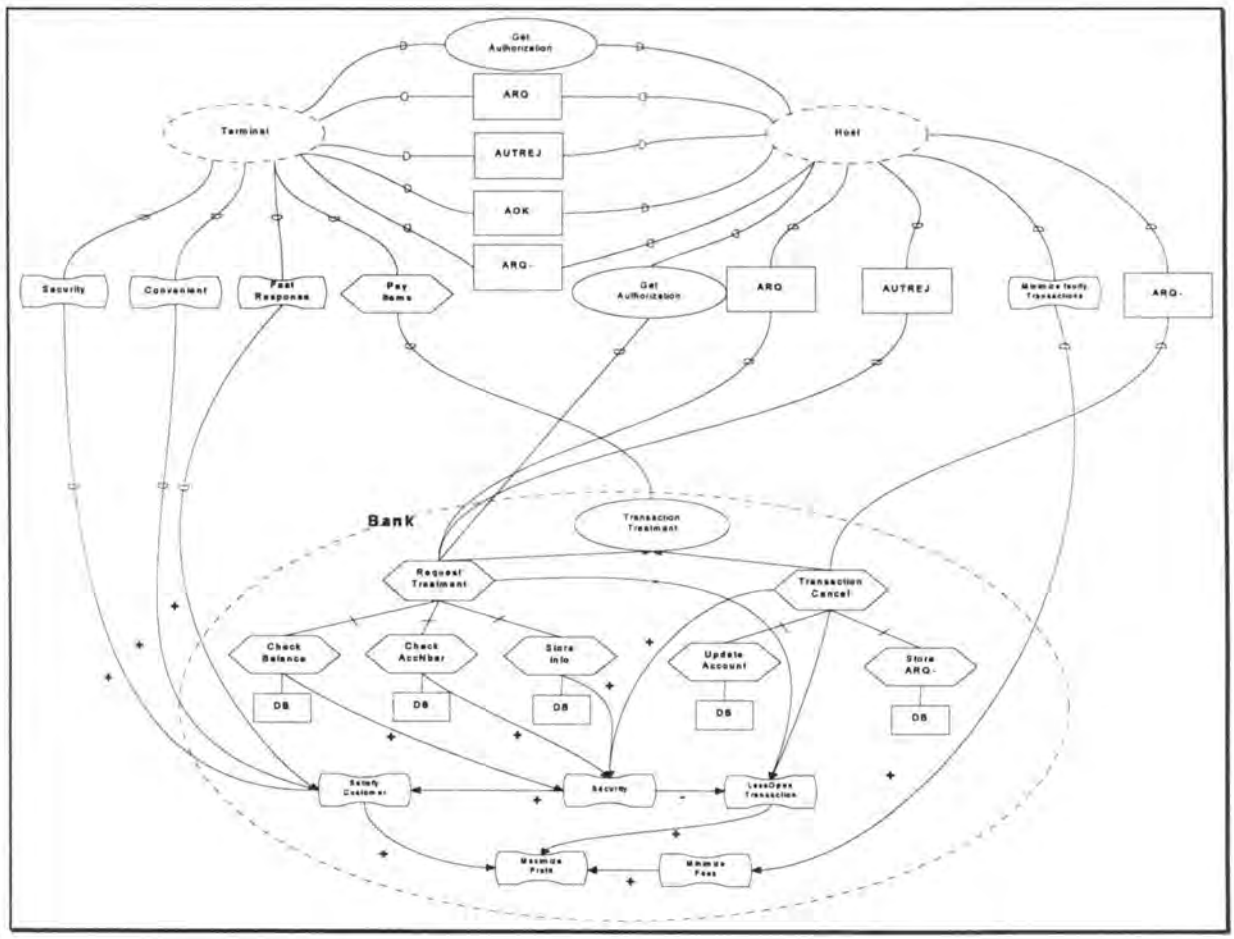


Figure 5.6. The Strategic Rationale model of the Bank actor

The aim of the *RequestTreatment* task is the same than in our *Four Messages Protocol* example. Lets however remark that the transaction amount is retrieved directly from the customer's account in the case where the transaction request has been accepted. In opposition to the 4MP example, the 2MP example does not require the confirmation of the transaction amount.

The *TransactionCancel* task is executed in the case where an ARQ- message is received. It consists in updating the customer's account balance and in storing the received Authorization Request Canceling (ARQ-) message. The customer's account is however only updated if the transaction amount has been previously retrieved. This means that a corresponding ARQ message has to be received before the reception of the ARQ- message by the bank and that the request has been accepted by the bank.

Section 3. The Albert specification of the Two Messages Protocol example

The Types

BASIC TYPES

CARD
PIN
DATE
TIME
DEBIT
REF_H
REF_BK

CONSTRUCTED TYPES

STATUS_H = ENUM ['Transaction Requested', 'Transaction Accepted', 'Transaction Refused',
'Transaction Cancelled', 'TimeOut']
STATUS_BK = ENUM['Transaction Accepted', 'Transaction Refused', 'Transaction Cancelled']
REASON_H = ENUM['', 'Invalid PinCode', 'Previously Cancelled']
REASON_BK = ENUM['', 'Amount not Covered', 'Previously Cancelled']

ARQ = CP [Card : CARD ; Price : INTEGER ; Pin : PIN ; Date : DATE ; Time : TIME]
REP = CP [Date : DATE ; Time : TIME ; Card : CARD ; Response : STRING ; Reason : STRING]
AOK = CP [Ref : INTEGER, Date : DATE, Time : TIME, Card : CARD]
ARQ- = CP [Card : CARD ; Price : INTEGER ; Pin : PIN ; Date : DATE ; Time : TIME ; Ref : INTEGER]
AUTREJ = CP [Host_ref : REF_H ; Response : STRING ; Reason : STRING]
REQUEST = CP [Debit_nber : DEBIT ; Price : INTEGER ; Date : DATE ; Time : TIME ; Host_ref : REF_H]
TRANS_H = CP [Card : CARD ; Price : INTEGER ; Date : DATE ; Time : TIME ; Status : STATUS_H;
Reason : REASON_H ; Term_id : TERMINAL]
TRANS_BK = CP [Debit_nber : DEBIT ; Price : INTEGER ; Date : DATE ; Time : TIME ;
Status : STATUS_BK ; Reason : REASON_BK, Host_ref : REF_H]

OPERATIONS

a. The Terminal

The Declaration of the Terminal agent

DECLARATIONS

STATE COMPONENTS

Memory table-of ARQ indexed-by INTEGER
TimeOutPeriod instance-of TIME
SendArq-Frequency instance-of TIME
Available instance-of BOOLEAN

ACTIONS

ScanCard(card) : the action of scanning the customer's debit card *card* through the terminal's cardreader

ScanCard(CARD)

EnterPrice(price) : the action of entering the transaction amount *price* by using the terminal's keyboard
EnterPrice(INTEGER)
EnterPin(pin) : the action of entering the customer's secret PIN Code *pin* by using the terminal's keyboard
EnterPin(PIN)
EnterTime(date, time) : the action of entering the transaction date *date* and time *time*
EnterTime(DATE, TIME)
SendMsgReq(arq, term_id) : the action of sending the transaction request *arq* and the terminal's identification code *term_id* to the Host
SendMsgReq(ARQ, TERMINAL)→ HOST
DisplayMsg(msg) : the action of displaying a message *msg* on the terminal's LCD display
DisplayMsg(STRING)
ResetTerminal : the action of freeing the terminal
ResetTerminal
StoreArq-(i, arq) : the action of storing an ARQ message *arq* into the terminal's memory at position *i*
StoreArq-(INTEGER, ARQ)
CreateArq-(i, arq-) : the action of creating an ARQ- message *arq-* by using the informations located at position *i* in the terminal's memory
CreateArq-(INTEGER, ARQ-)
SendMsgArq-(arq-, term_id) : the action of sending an ARQ- message *arq-* and the terminal's identification code *term_id* to the Host
SendMsgArq-(ARQ-, TERMINAL) → HOST
Remove(i) : the action of removing a stored record located at position *i* from the terminal's memory
Remove(INTEGER)

The Constraints of the Terminal agent

BASIC CONSTRAINTS

DERIVED COMPONENTS

INITIAL VALUATION

Memory[i] := undef
Available := true

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

In-Dom (Memory, arq) ⇒ ¬ SomeF(In-Dom (Memory, arq))

ACTION COMPOSITION

{ScanCard, EnterPrice, EnterPin, EnterTime, SendRequest, Envoi_OK, Envoi_KO, SendMsgReq, Response, Timeout, Host.SendMsgResponse, ResponseTreatment, DisplayMsg, FreeTerminal , ResetTerminal, StoreArq-, SendArq-, CreateArq-, SendMsgArq-, Host.SendMsgAok, AokTreatment, Remove}

Request ↔ ScanCard(card.arq) <> EnterPrice(price.arq) <> EnterPin(pin.arq) <> EnterTime(date.arq, time.arq) <> SendRequest(arq)

SendRequest(arq) ↔ (Envoi_OK(arq) ⊕ Envoi_KO(arq))

Envoi_OK(arq) ↔ SendMsgReq(arq, term_id) <> Response(arq)

Envoi_KO(arq) ↔ SendMsgReq(arq, term_id) <> Timeout(arq)

Response(arq) ↔ Host.SendMsgResponse(rep, term_id) <> (ResponseTreatment(arq, rep) ⊕ dac)

ResponseTreatment(arq, rep) ↔ DisplayMsg(response.rep + reason.rep) <> FreeTerminal

FreeTerminal ↔ DisplayMsg('Available') <> ResetTerminal

Timeout(arq) ↔ DisplayMsg('Transaction Cancelled') <> StoreArq-(i, arq) <> FreeTerminal

SendArq-(i) ↔ CreateArq-(i, arq-) <> SendMsgArq-(arq-, term_id)

AokReception ↔ Host.SendMsgAok(aok, term_id) <> (AokTreatment(aok) ⊕ dac)

AokTreatment(aok) ↔ Remove(ref.aok)

ACTION DURATION

|Envoi_OK(arq)| ≤ TimeOutPeriod
|Envoi_KO(arq)| > TimeOutPeriod

OPERATIONAL CONSTRAINTS

PRECONDITION

Request : Available
StoreArq-(i, _) : Memory[i] = undef
ResponseTreatment(arq, rep) : Date.arq = Date.rep ∧ Time.arq = Time.rep ∧ Card.arq = Card.rep ∧ Available
AokTreatment(aok) : Date.aok = Date.Memory[ref.aok] ∧ Time.aok = Time.Memory[ref.aok] ∧ Card.aok = Card.Memory[ref.aok]

EFFECTS OF ACTIONS

StoreArq-(i, arq) : []
Memory[i] := Card.arq, Price.arq, Pin.arq, Date.arq, Time.arq
Remove(ref.aok) : []
Memory[ref.aok] := undef
ResetTerminal : []
Available := true
ScanCard(card.arq) : []
Available := false

TRIGGERINGS

Memory[i] ≠ undef / SendArq-Frequency → SendArq-(i)

COOPERATION CONSTRAINTS

ACTION PERCEPTION

STATE PERCEPTION

ACTION INFORMATION

XK(SendMsgReq(arq, term_id).Host / true)
XK(SendMsgArq-(arq-, term_id).Host / true)

STATE INFORMATION

The Declaration associated with the TERMINAL agent

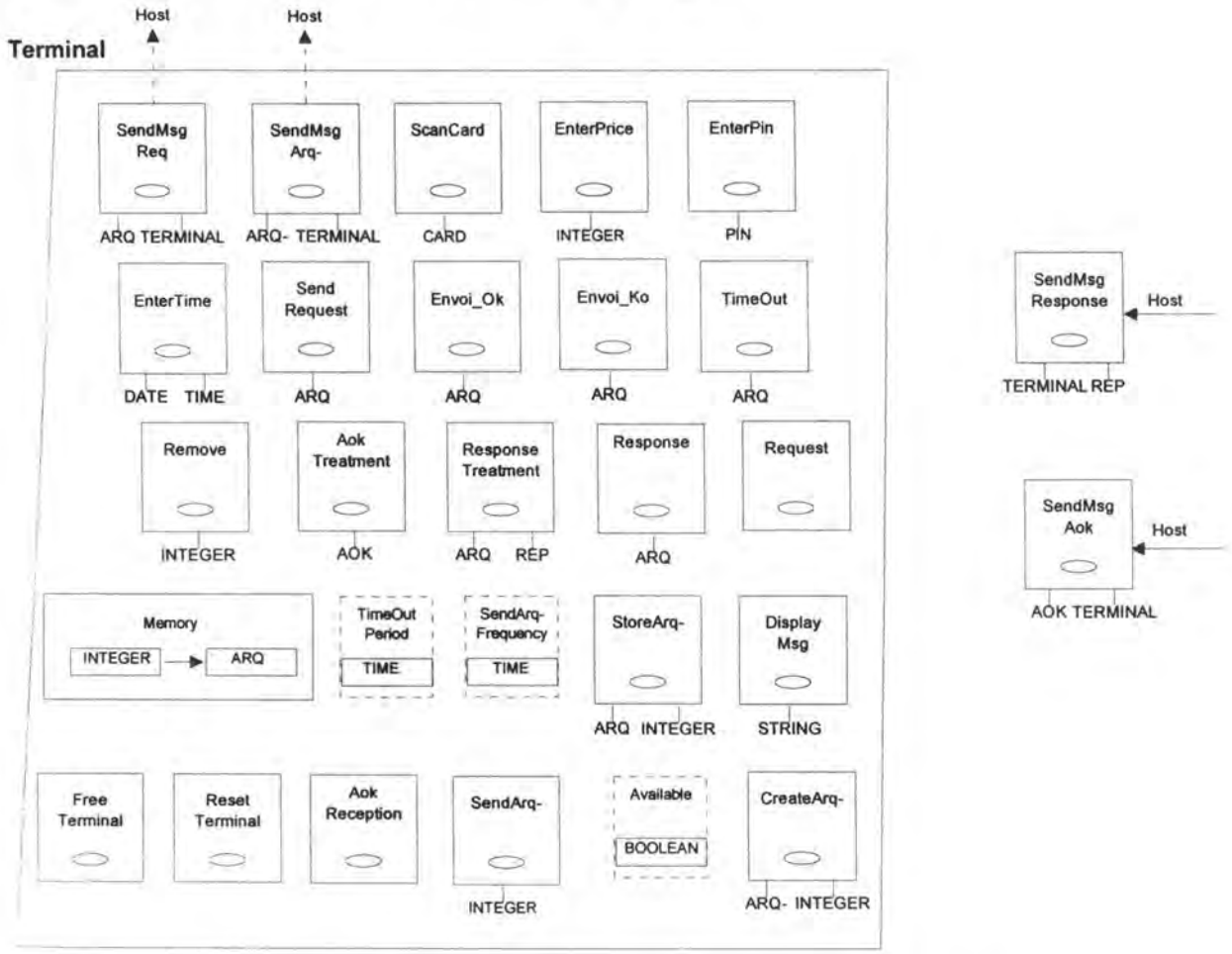


Figure 5.7. The Graphical Declaration of the Terminal agent

b. The Host agent

The Declaration of the Host agent

DECLARATIONS
STATE COMPONENTS

TimeOutPeriod instance-of TIME
Pin_Ok instance of BOOLEAN
Codes table-of PIN indexed-by CARD
DebitCard table-of DEBIT indexed-by CARD
Transactions table-of TRANS_H indexed-by REF_H

ACTIONS

CheckPin(card, pin) : the action of checking if the entered Pin Code *pin* corresponds to the received card Number *card*

CheckPin(CARD, PIN)

Search_AcNber(card, debit_nber) : the action of searching the account number *debit_nber* belonging to the received card number *card*

Search_AcNber(CARD, DEBIT)

StoreValid(arq, term_id, host_ref) : the action of recording a valid transaction request *arq* as well as

the terminal's identification code *term_id* at position *host_ref* into the transactions table

StoreValid(ARQ, TERMINAL, REF_H)

CreateRequest(debit_nber, host_ref, request) : the action of creating a request message *request* based on the customer's account number *debit_nber* as well as different informations previously stored in the transactions table at position *host_ref*

CreateRequest(DEBIT, REF_H, REQUEST)

FindOutBank(debit_nber, bank_id) : the action of finding out the identification code *bank_id* of the bank to which the request message is forwarded.

FindOutBank(DEBIT, BANK)

SendRequest(request, bank_id) : the action of forwarding the customer's transformed transaction request *request* to the customer's bank identified by *bank_id*

SendRequest(REQUEST, BANK)→BANK

UpdateTimeOut(host_ref) : the action of recording the occurrence of a Timeout by updating the transaction informations of the transaction located at position *host_ref* in the transaction table.

UpdateTimeOut(REF_H)

StoreInvalid(arq, term_id, host_ref) : the action of recording an invalid received transaction request *arq* as well as the terminal's identification code *term_id* into the transactions table at position *host_ref*

StoreInvalid(ARQ, TERMINAL, REF_H)

CreateReject(arq, rep) : the action of creating a negative response message *rep* based on an invalid received transaction request *arq*

CreateReject(ARQ, REP)

SendMsgResponse(rep, term_id) : the action of forwarding the response *rep* to the terminal identified by *term_id*

SendMsgResponse(REP, TERMINAL) →TERMINAL

UpdateTrans(autrej) : the action of updating the transactions table after the reception of the bank's response *autrej*

UpdateTrans(AUTREJ)

CreateRep(autrej, rep, term_id) : the action of creating a response message *rep* containing the bank's decision whether the transaction request has been accepted or not. The needed informations are contained in the received *autrej* message as well as in the transaction table.

CreateRep(REF_H, REP, TERMINAL)

StorePrevCancelled(arq, term_id, host_ref) : the action of recording a received transaction request *arq* that has previously been cancelled as well as the terminal's identification code *term_id* into the transaction table at position *host_ref.*

StorePrevCancelled(ARQ, TERMINAL, REF_H)

StoreArq-(arq-, term_id, host_ref) : the action of recording the reception of an ARQ- message *arq-* as well as the terminal's identification code *term_id* at position *host_ref* in the transaction table.

StoreArq-(ARQ-, TERMINAL, REF_H)

StoreArqNotExist(arq-, term_id, host_ref) : the action of recording the reception of an ARQ- message *arq-* as well as the terminal's identification code *term_id* in the case where the ARQ- message arrives before the corresponding ARQ message

StoreArqNotExist(ARQ-, TERMINAL, REF_H)

CreateArq-(host_ref, debit_nber, cancel) : the action of creating an ARQ- message *cancel* based on the customer's account number *debit_nber* as well as different informations previously stored in the transaction table at position *host_ref*

CreateArq-(REF_H, DEBIT, REQUEST)
SendArq-(cancel, bank_id) : the action of sending an ARQ- message *cancel* to the customer's bank identified by *bank_id*
SendArq-(REQUEST, BANK) →BANK
CreateAok(arq-, aok) : the action of creating an AOK message *aok* based on a received ARQ- message *arq-* CreateAok(ARQ-, AOK)
SendMsgAok (aok, term_id) : the action of sending an AOK message *aok* to the terminal identified by *term_id*
SendMsgAok (AOK, TERMINAL) →TERMINAL

The Constraints of the Host agent

BASIC CONSTRAINTS
DERIVED COMPONENTS
INITIAL VALUATION
Transactions[i] := undef

DECLARATIVE CONSTRAINTS
STATE BEHAVIOUR

ACTION COMPOSITION

{ARQ-Invalid, Arq-Treatment, ARQ-Valid, ArqNotExist, AutrejTreatment, Bank.EnvoiMsg, CheckPin, CreateAok, CreateArq-, CreateReject, CreateRep, CreateRequest, EnvoiKo, EnvoiOk, ExistArq, False_Pin, FindOutBank, Forward, InformBankArq-, InformCancelled, InformTerm, NotYetCancelled, PreviouslyCancelled, ResponseTreatment, Search_AcNber, SendArq-, SendMsgAok, SendMsgResponse, SendRequest, StoreArq-, StoreArqNotExist, StoreInvalid, StorePrevCancelled, StoreValid, Terminal.SendMsgArq-, Terminal.SendMsgReq, TimeOut, UpdateTimeOut, UpdateTrans, Valid_Pin}

RequestTreatment ↔ Terminal.SendMsgReq(arq, term_id) <> (NotYetCancelled(arq, term_id) ⊕ PreviouslyCancelled(arq, term_id))

NotYetCancelled(arq, term_id) ↔ CheckPin(card.arq, pin.arq) <> (Valid_Pin(arq, term_id) ⊕ False_Pin(arq, term_id))

Valid_Pin(arq, term_id) ↔ Search_AcNber(card.arq, debit_nber) <> StoreValid(arq, term_id, host_ref) <> CreateRequest(debit_nber, host_ref, request) <> FindOutBank(debit_nber, bank_id) <> Forward(request, bank_id)

Forward(request, bank_id) ↔ (EnvoiOk(request, bank_id) ⊕ EnvoiKo(request, bank_id))

EnvoiOk(request, bank_id) ↔ SendRequest(request, bank_id) <> ResponseTreatment

EnvoiKo(request, bank_id) ↔ SendRequest(request, bank_id) <> TimeOut(host_ref.request)

TimeOut(host_ref.request) ↔ UpdateTimeOut(host_ref.request)

False_Pin(arq, term_id) ↔ StoreInvalid(arq, term_id, host_ref) <> CreateReject(arq, rep) <> SendMsgResponse(rep, term_id)

ResponseTreatment ↔ Bank.EnvoiMsg(autrej, bank_id) <> (AutrejTreatment(autrej) ⊕ dac)

AutrejTreatment(autrej) ↔ UpdateTrans(autrej) <> CreateRep(host_ref.autrej, rep, term_id) <>

SendMsgResponse (rep, term_id)

PreviouslyCancelled(arq, term_id) ↔ StorePrevCancelled(arq, term_id, host_ref) <>
CheckPin(card.arq, pin.arq) <>
(InformCancelled(host_ref) ⊕ dac)

InformCancelled(host_ref) ↔ Search_AcNber(card.Transactions[host_ref], debit_nber) <>
CreateRequest(debit_nber, host_ref, request) <>
FindOutBank(debit_nber, bank_id) <> SendRequest(request, bank_id)

CancelTreatment ↔ Terminal.SendMsgArq-(arq-, term_id) <> CheckPin(card.arq-, pin.arq-) <>
(ARQ-Valid(arq-, term_id) ⊕ ARQ-Invalid(arq-, term_id))

Arq-Valid(arq-, term_id) ↔ (ExistArq(arq-, term_id) ⊕ ArqNotExist(arq-, term_id))

Arq-Treatment(arq-, term_id) ↔ StoreArq-(arq-, term_id, host_ref) <> InformBankArq-(host_ref)

ExistArq(arq-, term_id) ↔ (Arq-Treatment(arq-, term_id) ⊕ dac) <> InformTerm(arq-, term_id)

ArqNotExist(arq-, term_id) ↔ StoreArqNotExist(arq-, term_id, host_ref) <> InformBankArq-(host_ref)

InformBankArq-(host_ref) ↔ Search_AcNber(card.Transactions[host_ref], debit_nber) <>
CreateArq-(host_ref, debit_nber, cancel) <>
FindOutBank(debit_nber, bank_id) <>
SendArq-(cancel, bank_id)

ARQ-Invalid(arq-, term_id) ↔ InformTerm(arq-, term_id)

InformTerm(arq-, term_id) ↔ CreateAok(arq-, aok) <> SendMsgAok (aok, term_id)

ACTION DURATION

|EnvoiOk(request, bank_id)| ≤ TimeOutPeriod
|EnvoiKo(request, bank_id)| > TimeOutPeriod

OPERATIONAL CONSTRAINTS

PRECONDITION

NotYetCancelled(arq, term_id) : In-Dom (Transactions, trans)

with Date.arq = Date.trans
Time.arq = Time.trans
Card.arq = Card.trans
Status.trans ≠ 'Transaction Cancelled'

The NotYetCancelled action may only be executed in the case where the Transactions table does not contain a transaction *trans* corresponding to the received ARQ message *arq* and whose status is equal to 'Transaction cancelled'. In other words, the NotYetCancelled action is executed if the received ARQ message *arq* has not yet been cancelled by a corresponding ARQ- message.

PreviouslyCancelled(arq, term_id) : In-Dom (Transactions, trans)

with Date.arq = Date.trans
Time.arq = Time.trans
Card.arq = Card.trans

Status.trans = 'Transaction Cancelled'

The Host can only execute the PreviouslyCancelled action, in the case where the transaction table does not contain a transaction record *trans* corresponding to the received ARQ message *arq and* whose status is equal to 'Transaction Cancelled'. The PreviouslyCancelled action can only be executed if the received ARQ message *arq* has been previously cancelled.

Valid_Pin(arq, term_id) : Pin_Ok

The Valid_Pin action can only be executed if the customer has entered a valid Pin code i.e. the value of the Pin_Ok state component is true

False_Pin(arq, term_id) : ¬ Pin_Ok

The False_Pin action can not be executed if the customer has entered a valid Pin code

StoreValid(arq, term_id, host_ref) : Transactions[host_ref] = undef
StoreInvalid(arq, term_id, host_ref) : Transactions[host_ref] = undef
StorePrevCancelled(arq, term_id, host_ref) : Transactions[host_ref] = undef
StoreArq-(arq-, term_id, host_ref) : Transactions[host_ref] = undef
StoreArqNotExist(arq-, term_id, host_ref) : Transactions[host_ref] = undef

A new record can only be added to the Transactions table (at position host_ref), if the position host_ref does not contain a previously stored record.

InformCancelled(host_ref) : Pin_Ok

The Host forwards the received ARQ message arq to the Bank agent (i.e. execute the InformCancelled action) in the case where the customer has entered a valid Pin Code

Arq-Valid(arq-, term_id) : Pin_Ok

The execution of the ARQ-Valid action requires the entering of a valid Pin Code.

Arq-Invalid(arq-, term_id) : ¬Pin_Ok

The Host executes the ARQ-Invalid action in the case where the customer has entered an invalid Pin Code.

Arq-Treatment(arq-, term_id) : ¬ In-Dom (Transactions, trans)

with Date.arq- = Date.trans
Time.arq- = Time.trans
Card.arq- = Card.trans
Status.trans = 'Transaction Cancelled'

The Host may only execute the Arq-Treatment action in the case where an ARQ message has previously been received corresponding the received ARQ- message arq- .

ExistArq(arq-, term_id) : In-Dom (Transactions, a)

with Date.arq- = Date.a
Time.arq- = Time.a
Card.arq- = Card.a
(Status.a =' Transaction Requested' ∨
Status.a =' Transaction Accepted ' ∨
Status.a =' Transaction Refused')

The ExistArq action is executed in the case where the ARQ message corresponding to the received ARQ-message has been epreviously treated.

ArqNotExist(arq-, term_id) : ¬ (In-Dom (Transactions, a))

with Date.arq- = Date.a
Time.arq- = Time.a
Card.arq- = Card.a
(Status.a =' Transaction Requested' ∨
Status.a =' Transaction Accepted ' ∨
Status.a =' Transaction Refused')

The ArqNotExist action is executed in the case where the ARQ- message arq- has been received before its corresponding ARQ message.

AutrejTreatment(autrej) : Status.Transactions[host_ref.autrej] ≠ 'TimeOut'

The execution of the AutrejTreatment action requires the non occurrence of a TimeOut at the Host's level for the corresponding transaction request.

EFFECTS OF ACTIONS

CheckPin(card.arq, pin.arq) : []
Pin_Ok := (Pin.arq = Codes[Card.arq])

StoreValid(arq, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq, Date.arq, Time.arq,
'Transaction Requested', Term_id

StoreInvalid(arq, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq, Date.arq, Time.arq,
'Transaction Refused', 'Invalid PinCode',
Term_id

StorePrevCancelled(arq, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq, Date.arq,
Time.arq, Transaction Refused',
'Previously Cancelled', Term_id

StoreArq-(arq-, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq, Date.arq, Time.arq,
'Transaction Cancelled', '', Term_id

StoreArqNotExist(arq-, term_id, host_ref) : []
Transactions[host_ref] := Card.arq, Price.arq, Date.arq, Time.arq,
'Transaction Cancelled', '', Term_id

UpdateTrans(autrej) : []
Status.Transactions[host_ref] := Response.autrej
Reason.Transactions[host_ref] := Reason.autrej

UpdateTimeOut(host_ref.request) : []
Status.Transactions[host_ref.request] := ' TimeOut'

TRIGGERINGS

COOPERATION CONSTRAINTS

ACTION PERCEPTION

XK(Bank.EnvoiMsg(autrej, bank_id) / true)

STATE PERCEPTION

ACTION INFORMATION

XK(SendRequest(request, bank_id).Bank / true)
XK(SendMsgResponse(rep, term_id).Terminal / true)
XK SendArq-(cancel, bank_id).Terminal / true)
XK(SendMsgConf(conf_h, bank_id).Bank / true)

STATE INFORMATION

The Declaration associated with the HOST agent

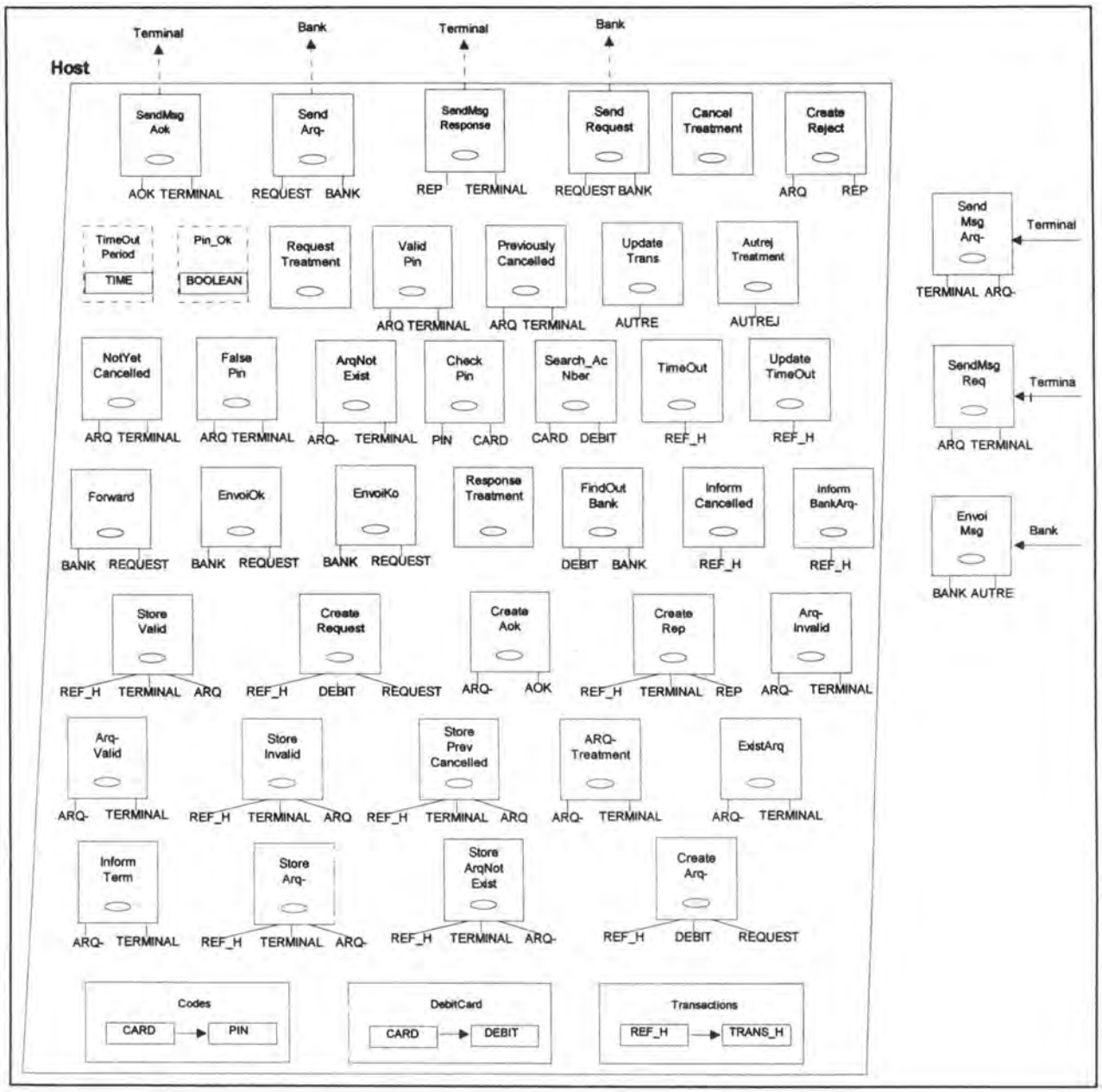


Figure 5.8. The Graphical Declaration of the Host agent

c. The Bank Agent

The Declaration of the Bank Agent

DECLARATIONS
STATE COMPONENTS

Transactions table-of TRANS_BK indexed-by REF_BK
Accounts table-of INTEGER indexed-by DEBIT
BalanceOk instance-of BOOLEAN

ACTIONS

Check_Balance(request) : the action of checking the customer's balance contained in the received request *request*

Check_Balance(REQUEST)

StoreAccepted(request, bank_ref) : the action of storing the accepted transaction request *request* into the transaction table at position *bank_ref*

StoreAccepted(REQUEST, REF_BK)

UpdateAcc(debit_nber, price) : the action of retrieving the transaction amount *price* from the customer's bank account *debit_nber*

UpdateAcc(DEBIT, PRICE)

CreateRep(bank_ref, autrej) : the action of creating a response message *autrej* containing the bank's response based on the informations contained in the transaction table at position *bank_ref*

CreateRep(REF_BK, AUTREJ)

EnvoiMsg(autrej, bank_id) : the action of sending the response message *autrej* and the bank's identification code *bank_id* to the Host agent

EnvoiMsg(AUTREJ, BANK)→HOST

StoreRefused(request, bank_ref) : the action of storing the refused transaction request *request* into the transaction table at position *bank_ref*

StoreRefused(REQUEST, REF_BK)

StoreCancelled(request, bank_ref) : the action of recording a ARQ- message *cancel* arriving before its corresponding ARQ message at position *bank_ref* into the transaction table

StoreCancelled(REQUEST, REF_BK)

StoreRequest(cancel, bank_ref) : the action of recording a received transaction Canceling message *cancel* into the transaction table at position *bank_ref*

StoreRequest(REQUEST, REF_BK)

UpdateARQ-(cancel) : the action of updating the customer's bank account specified in the received ARQ- message *cancel*

UpdateARQ-(REQUEST)

The Constraints of the Bank agent

BASIC CONSTRAINTS

DERIVED COMPONENTS

INITIAL VALUATION

Transactions[i] := undef

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

ACTION COMPOSITION

{Host.SendRequestNotYetCancelled, SpecialTreatment, Check_Balance, Accepted, Refused, StoreAccepted, UpdateAcc, StoreRefused, CreateRep, EnvoiMsg, StoreCancelled, Host.SendArq-, StoreRequest, UpdateArq-}

Request ↔ Host.SendRequest(request, bank_id) <> (NotYetCancelled(request) ⊕ SpecialTreatment(request))

NotYetCancelled(request) ↔ Check_Balance(request) <> (Accepted(request) ⊕ Refused(request))

Accepted(request) ↔ StoreAccepted(request, bank_ref) <> UpdateAcc(debit_nber.request, price.request) <> CreateRep(bank_ref, autrej) <> EnvoiMsg(autrej, bank_id)

Refused(request) ↔ StoreRefused(request, bank_ref) <> CreateRep(bank_ref, autrej) <> EnvoiMsg(autrej, bank_id)

SpecialTreatment(request) ↔ StoreCancelled(request, bank_ref)

RequestArq- ↔ Host.SendArq-(cancel, bank_id) <> StoreRequest(cancel, bank_ref) <>
(UpdateArq-(cancel) ⊕ dac)

ACTION DURATION

COOPERATION CONSTRAINTS

PRECONDITION

SpecialTreatment(request) : (In-Dom (Transactions, tr))

with Date.request = Date.tr
Time.request = Time.tr
Debit_nber.request = Debit_nber.tr
Status.tr = 'Transaction Cancelled'

NotYetCancelled (request) : ¬ (In-Dom (Transactions, tr))

with Date.request = Date.tr
Time.request = Time.tr
Debit_nber.request = Debit_nber.tr
Status.tr = 'Transaction Cancelled'

Accepted(request) : BalanceOk

Refused(request) : ¬ BalanceOk

UpdateArq-(cancel) / In-Dom (Transactions, tr)
with Date.cancel = Date.tr ∧
Time. cancel = Time.tr ∧
Debit_nber.cancel = Debit_nber.tr ∧
Status.tr = 'Transaction Accepted'

StoreAccepted(request, bank_ref) : Transactions[bank_ref] = undef

StoreRefused(request, bank_ref) : Transactions[bank_ref] = undef

StoreCancelled(request, bank_ref) : Transactions[bank_ref] = undef

StoreRequest(cancel, bank_ref) : Transactions[bank_ref] = undef

EFFECTS OF ACTIONS

Check_Balance(request) : []
BalanceOk := (Accounts[debit_nber.request] - Price.request) ≥ 0

StoreAccepted(request, bank_ref) : []
Transactions[bank_ref] := Debit_nber.request, Price.request, Date.request,
Time.request, Host_ref.request,
'Transaction Accepted'

StoreAccepted(request, bank_ref) : []
Transactions[bank_ref] := Debit_nber.request, Price.request,
Date.request, Time.request, Host_ref.request,
'Transaction Refused',
'Amount not Covered'

StoreCancelled(request, bank_ref) : []
Transactions[bank_ref] := Debit_nber.cancel, Price.cancel, Date.request,
Time.request, Host_ref.cancel,
'Transaction Refused', 'Previously Cancelled'

StoreRequest(cancel) : []
Transactions[bank_ref] := Debit_nber.cancel, Price.cancel, Date.request,
Time.request, Host_ref.cancel, 'Transaction Cancelled'

UpdateAcc(debit_nber.request, price.request) : []
Accounts[debit_nber.request] := Accounts[debit_nber.request]
- Price.request

UpdateARQ-(cancel) : []
Accounts[debit_nber.cancel] := Accounts[debit_nber.cancel] + Price.cancel

TRIGGERINGS

COOPERATION CONSTRAINTS

ACTION PERCEPTION

XK(Host.SendRequest(request, bank_id) / bank_id = self)
XK(Host.SendArq-(cancel, bank_id) / bank_id = self)

STATE PERCEPTION

ACTION INFORMATION

XK(EnvoiMsg(autrej, bank_id).Host / true)

STATE INFORMATION

The Declaration associated with the BANK agent

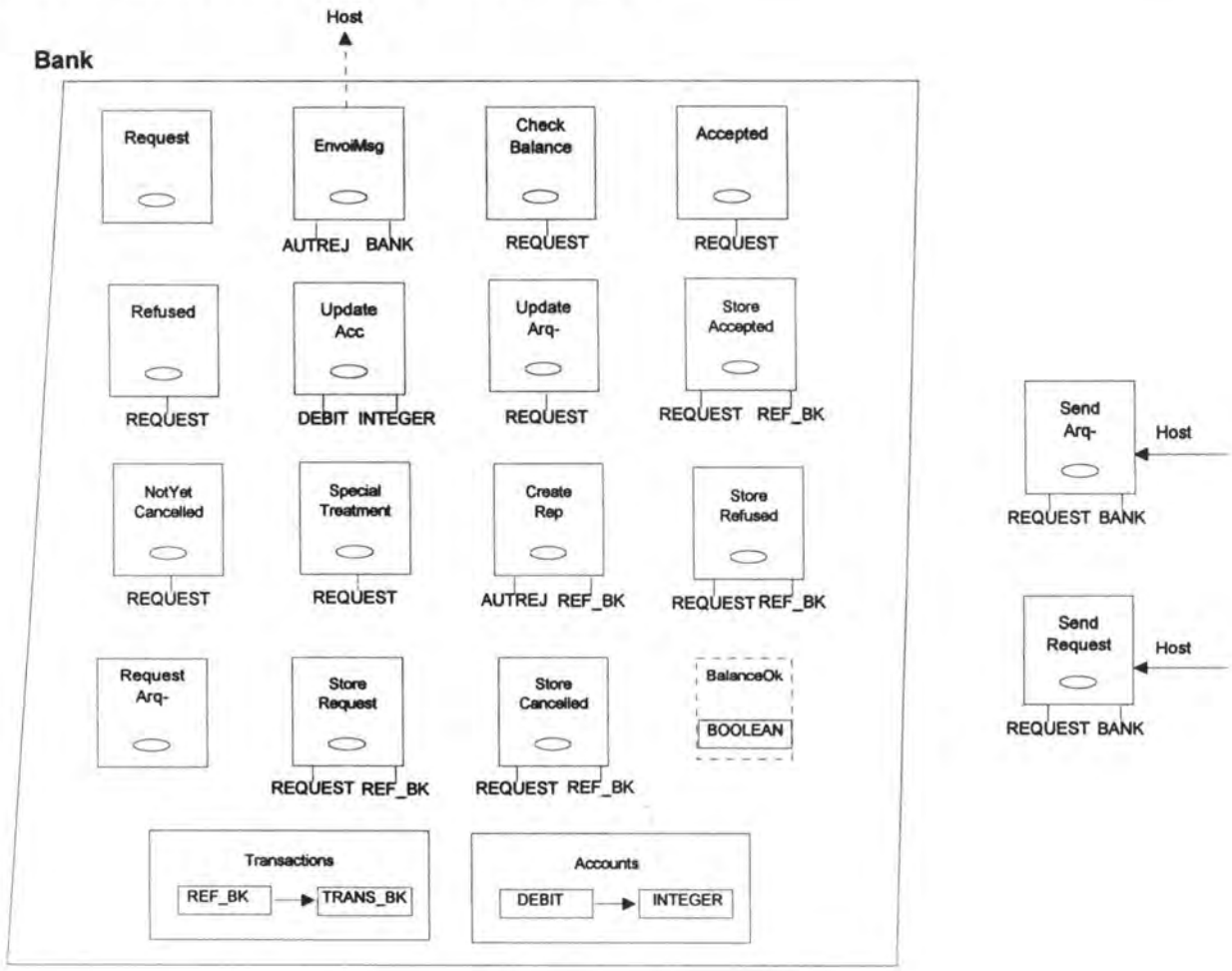


Figure 5.9. The Graphical Declaration of the Bank agent